Reparallelization and Migration of OpenMP Applications in Grid Environments

Reparallelisierung und Migration von OpenMP-Applikationen in Grid-Umgebungen

Der Technischen Fakultät der Universität Erlangen-Nürnberg zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von
Michael Klemm

Erlangen - 2009
Als Dissertation genehmigt von der Technischen Fakultät
der Universität Erlangen-Nürnberg.

Tag der Einreichung: 30.06.2008

Tag der Promotion: 17.10.2008

Dekan: Professor Dr.-Ing. habil. J. Huber

Berichterstatter: Professor Dr. M. Philippson

Professor Dr. U. Rüde
SI ENIM PLACET OPUS MEUM, GAUDEBO.

SI AUTEM NULLI PLACET: MEMET IPSUM TAMEN IUVAT QUOD FECI.
An author of a doctoral thesis is required to research and prepare it without the help of others. And this is what I did. However, there are various people that directly or indirectly supported me while working on my masterpiece. They all deserve my special thanks.

First of all, I would like to thank Professor Dr. Michael Philippsen for the opportunity to work on his staff on many interesting topics during these past few years. He always believed in my skills and supported my work by providing many useful comments on my research papers and my thesis. However, I will take the time to say that I will never join him for a bicycle tour ever again! Professor Dr. Rüde devoted his time to review my work and report on it. Professor Dr. Christian Stegmann and Professor Dr. Marc Stamminger completed the committee for my final examination. All members of the committee created a pleasant atmosphere for the examination and asked challenging questions.

Many thanks to my students Matthias Bezold, Benjamin Bieber, Georg Dotzler, Stephan Gabriel, and Christian Iwainsky. They contributed their passion, a lot of debugging, and countless lines of code to my work. I apologize for all the parties you missed, because you spent the night in the computer rooms keeping your hands busy on the keyboard.
Preface

Special thanks go to my colleagues Alexander Dreweke, Johannes Drexler, Dr. Bernd Hardung, Dr. Martin Jung, Christoph Neumann, Tobias Werth, and Marc Wörlein who were always present for an endless number of coffee breaks at our office as well as our “team meetings” in bars. Dominic Schell and Szilvia Zvada became close friends in rough times. Dr. Peter Wilke always had an open door for me in case of any problem or just a really bad mood. Dr. Jean-Christophe Beyler was a great co-worker on another project and working on research papers together was an unforgettable experience. Dr. Ulrike Ranger and Dr. Steffen Möller provided so many useful comments on the very first versions of the text.

My parents and my sisters were always there for me when I was in trouble and when I needed their help. Their fridge was always open for a hungry doctoral student who missed so many meals because of a paper submission deadline. Their support made my studies and my doctoral thesis possible.

Last but not least, I would like to thank my partner and true love Martina for her never-ending support and her understanding during the last seven years. She always put her own needs on the backburner to help me find as much time as possible for my work. All this would not have been possible without you. Thank you!

Michael Klemm,
Erlangen, 2009
Abstract

Grid computing opens up a new computing infrastructure and users gain access to a vast landscape of computing resources that are spread over the world and that are instantly accessible through the Internet. In the future, computational Grids are expected to deliver high-end computing power at the user’s finger tips.

Today, typical users of a computational Grid only exploit a single cluster and are faced with the job schedulers that assign computing resources to applications. Job schedulers expect a resource estimation to exclusively allocate parts of the computing system for the application. Users typically over-estimate the resource limits of their applications to avoid premature termination of the application by the job scheduler. Over-estimating resource limits has negative effects on both the clusters’ schedules as well as on the waiting time until an application eventually executes.

In this work, we present a solution that alleviates the need of estimating the resource limits for OpenMP applications. A reparallelization of OpenMP applications is automatically computed, when new computing resources become available or are withdrawn. The application can be migrated between clusters of the Grid if an allocated resource is about to be exceeded.

Programmers do not need to change the application to enable reparallelization and migration. Instead, our compiler transparently prepares the applications for reparallelization and migration. It adds code to the application to enable reparallelization and augments the application with a platform-independent, coordinated checkpointing algorithm for migration. A prototype implementation of a migration framework automatically discovers free resources and migrates the application to these resources. Measurements show that the overhead of reparallelization and migration is roughly 4%, which we consider a negligible cost compared to the gain of flexibility.
Kurzfassung


Mit der vorliegenden Arbeit wird eine Lösung präsentiert, welche überhöhte Schätzungen für OpenMP-Applikationen überflüssig macht. Sollten mehr Ressourcen zur Verfügung stehen oder der Applikation entzogen werden, so wird eine Reparallelisierung einer OpenMP-Applikation automatisch durchgeführt. Weiterhin kann eine Applikation zwischen Rechnerbündeln eines Grids migriert werden, sobald die momentan genutzte Ressource kurz vor der erzwungenen Freigabe steht.

Kurzfassung
Bibliographic Notes

Preliminary versions of parts of this work have also been published in the following research papers:


- Michael Klemm and Michael Philippsen: Reparallelisierung und Migration von OpenMP-Applikationen. In: *Mitteilungen — Gesellschaft*
Bibliographic Notes


The presented work was awarded the “Young Researchers Award 2007” of the special interest group on “Parallel-Algorithmen und Rechnerstrukturen” of the Gesellschaft für Informatik, e. V.
# Contents

1 Introduction
   1.1 Terminology .................................................. 6
   1.2 Classification of Parallel Applications ...................... 7
   1.3 Contributions ................................................. 10
   1.4 State of the Art ............................................... 11

2 OpenMP for Java
   2.1 State of the Art ............................................... 14
      2.1.1 Parallel Programming Models ............................. 14
      2.1.2 Language Extensions for Java .......................... 17
   2.2 Overview of OpenMP ........................................... 19
      2.2.1 History of OpenMP ....................................... 19
      2.2.2 Programming Model ...................................... 21
      2.2.3 Memory Consistency Model .............................. 23
   2.3 Adapting OpenMP to Java ..................................... 25
      2.3.1 Design Objectives ....................................... 26
      2.3.2 OpenMP/JavaPragma Syntax .............................. 26
      2.3.3 Constructs and Directives ................................ 28
      2.3.4 Runtime Support Library ................................ 30
      2.3.5 Determining the Number of OpenMP Threads .......... 33
      2.3.6 Changes and Extensions to OpenMP/Java Constructs ... 35
      2.3.7 Exception Handling ...................................... 43
      2.3.8 OpenMP/Java Memory Model .............................. 44
## Contents

5.5 Checkpointing a Single Process .................................. 164
  5.5.1 Single-thread Checkpointing ................................. 165
  5.5.2 Multi-thread Checkpointing ............................... 168
  5.5.3 Checkpointing the Static Data and the Heap .......... 170

5.6 Distributed Checkpointing Algorithm ............................. 172
  5.6.1 Introduction to Distributed Checkpointing .......... 173
  5.6.2 Distributed Checkpointing Algorithm .................. 175

5.7 Checkpointing the DSM Protocol State ............................ 180
  5.7.1 The Jackal DSM Protocol ............................... 180
  5.7.2 DSM State Checkpointing ............................... 183
  5.7.3 Changing the Number of Processes during Restoration 185
  5.7.4 Interaction with Reparallelization .................... 187

5.8 Impact of Checkpointing on Application Semantics ........... 189

5.9 Summary ......................................................... 191

6 Framework for Inter-cluster Migration ............................. 193
  6.1 State of the Art ............................................... 194
  6.2 Design Objectives ........................................... 195
  6.3 The OGRE Migration Framework ............................... 197
    6.3.1 Components ............................................ 197
    6.3.2 Application Life Cycle ................................ 200
  6.4 Migration Strategy ............................................ 202
    6.4.1 Quantifying the Performance of Clusters .......... 202
    6.4.2 Quantifying the Performance of Cluster Reservations . 203
    6.4.3 Benefit Function ...................................... 204
    6.4.4 Computing the Migration Plan ......................... 205
  6.5 Application Migration ........................................ 210
  6.6 Summary ......................................................... 211

7 Performance Evaluation ............................................. 213
  7.1 Microbenchmarks ............................................... 214
  7.2 Benchmarks ..................................................... 217
  7.3 OpenMP/Java .................................................. 221
    7.3.1 Thread Binding ........................................ 221
7.3.2 M:N Mapping ........................................ 224
7.4 Reparallelization ........................................ 228
7.5 Checkpointing .......................................... 235
7.6 Migration .............................................. 237
7.7 Summary ............................................... 239

8 Conclusions and Future Work ................................. 241
  8.1 Conclusions ........................................ 241
  8.2 Future Work ....................................... 244
List of Figures

1.1 Cumulative histogram: reserved versus used runtime. . . . . . 3
1.2 Distribution of the jobs sizes on the Woodcrest cluster. . . . . 4
1.3 Example schedule with reservation hole. . . . . . . . . . . . . 5
1.4 Classification scheme for parallel applications. . . . . . . . . 7
1.5 Classification scheme for malleable applications. . . . . . . . 9

2.1 Evolution of the OpenMP specifications over time. . . . . . 20
2.2 Example of a parallel reduction of array elements. . . . . . 21
2.3 Example distribution of a loop of length 20. . . . . . . . . . . 21
2.4 Schematic visualization of the fork/join programming model
(8 threads). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.5 OpenMP memory model: shared memory and thread private
memory. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.6 Java runtime support functions and additional functions for
object pooling. . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.7 Determining the number of threads for a parallel region. . . 34
2.8 Interface definition for custom object copying and example of
usage. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.9 Parallel iteration over a Java collection. . . . . . . . . . . . . 39
2.10 Interface definition for user-defined reduction operators and
usage example. . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.1 Classification scheme for OpenMP implementations. . . . . . 53
3.2 The Jackal compiler pipeline. . . . . . . . . . . . . . . . . . . 59
3.3 Architecture of an OpenMP compiler. . . . . . . . . . . . . . 61
3.4 Example rules of the OpenMP/Java grammar. . . . . . . . . . 63
List of Figures

3.5 Simplified AST representation of the sumArray example of Figure 2.2. .............................................. 64
3.6 Example of the AST transformation for parallel sections. .............................................................. 67
3.7 High-level LASM representation of the example in Figure 2.2. ....................................................... 69
3.8 Pseudo-code example of data-scoping arguments for OpenMP thunks. ............................................. 73
3.9 Pseudo-code example of the translation of OpenMP-parallel loops. .................................................. 75
3.10 Skeleton of the exception handler in JaMP’s executor threads. ....................................................... 77
3.11 Example execution of four OpenMP threads on different numbers of system units of execution. ........ 80
3.12 Example of a parallel region with barrier directive. .............. 81
3.13 Example for LUE splitting of a parallel region with one barrier. .......... 82
3.14 Counter example for LUE splitting of a parallel region with two barriers. ..................................... 83
3.15 Example of optimizations for LUE barriers. .......................................................... 85
3.16 Functor and closure to implement OpenMP/Java thunks. ........... 89
3.17 Pseudo-code generated for sumArray in Figure 2.2. ......... 92
3.18 Example of an asynchronous computation of an array sum. .... 97

4.1 Example of a manual parallelization of the array sum example (main program). ................................ 111
4.2 Example of a manual parallelization of the array sum example (thread and reducer). ................. 112
4.3 Iteration space of a parallel loop with iterations, chunks, and partial chunks. ................................ 115
4.4 Example of the adjust directive to define adjustment points. 116
4.5 Reassignment example for a static loop schedule. .......... 121
4.6 Variable context at an adjustment point. .......... 129
4.7 Example for nested parallelism in OpenMP/Java. ......... 130
4.8 Dependency to the thread ID. .......................... 132
4.9 Corrected example of Figure 4.8. .................. 132
4.10 Example code showing semantic issues with reparallelization. 133
4.11 Generic skeleton of the code templates for adjustment points. 138
4.12 Code templates for adjustment points in work-sharing constructs. ............................................. 139
4.13 Code templates for adjustment points in parallel regions. .. 140
4.14 Reparallelization barrier at work-sharing constructs. ....... 143
5.1 Example method with three variables. .......................... 157
5.2 Allocation of variables in the example of Figure 5.1 on IA-32 and EM64T. ...................................................... 157
5.3 Example code with live variables to illustrate the creation of UDS. ................................................................. 163
5.4 Checkpointer and an uncheckpointer for the example code in Figure 5.3. ............................................................. 166
5.5 Pseudo-code for unwinding and checkpointing a single call stack. 167
5.6 Pseudo-code for restoring a single call stack. ..................... 167
5.7 Orphan message between P1 and P2. .............................. 174
5.8 Lost message between P1 and P2. ................................. 174
5.9 Recovery line of a checkpoint. ....................................... 175
5.10 Visualization of the distributed checkpointing algorithm. ...... 176
5.11 Coordinated checkpointing algorithm for the Jackal DSM. ... 178
5.12 Application heaps for Jackal applications compiled for DSM usage. ................................................................. 181
5.13 A function before (Figure 5.13(a)) and after the insertion of an access check (Figure 5.13(b)). ............................... 182
5.14 Splitting three processes into six processes at restart. ......... 186
5.15 Merging six process images into three process images. ....... 187
6.1 Architecture of the migration framework. ....................... 198
6.2 States of the application life cycle in OGRE. .................... 201
6.3 Cuboid quantifying the computing power of a cluster reservation. ................................................................. 203
6.4 Example of incoming bids containing reservation holes. ...... 206
6.5 Graph representation of the bids in Figure 6.4. ................... 207
6.6 Longest path algorithm. ............................................. 209
6.7 Example of a job script for submission. ........................... 210
7.1 Runtime costs of OpenMP/Java constructs and directives. ... 216
7.2 Runtime costs of scheduling types in OpenMP/Java. .......... 218
7.3 Speed-ups of the OpenMP/Java benchmarks (thread binding). 222
7.4 Speed-ups of the OpenMP-parallel benchmarks (m:n mapping). 225
7.5 Speed-ups of the OpenMP-parallel benchmarks with adjustment points added. ................................................... 231
7.6 Reparallelization of SOR (runtime per iteration). .............. 233
7.7 Reparallelization of LBM (runtime per iteration). ............ 234
List of Figures

7.8 Speed-ups of the OpenMP/Java benchmarks with checkpoint support added. ............................................. 236
7.9 Continuous migration of LBM between Woodcrest and DAS3. 238
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>OpenMP constructs and directives supported by OpenMP/Java.</td>
<td>48</td>
</tr>
<tr>
<td>2.2</td>
<td>OpenMP clauses supported by OpenMP/Java.</td>
<td>49</td>
</tr>
<tr>
<td>3.1</td>
<td>Classification of OpenMP implementations.</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of malleability in different programming models.</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td>Classification of checkpointing techniques.</td>
<td>146</td>
</tr>
<tr>
<td>5.2</td>
<td>Classification of checkpointing packages.</td>
<td>147</td>
</tr>
<tr>
<td>7.1</td>
<td>Data sizes used for the benchmarks.</td>
<td>219</td>
</tr>
<tr>
<td>7.2</td>
<td>Overheads of JaMP’s $m:n$ binding with respect to the thread binding.</td>
<td>227</td>
</tr>
<tr>
<td>7.3</td>
<td>Overheads of adjustment points and checkpointing added.</td>
<td>229</td>
</tr>
<tr>
<td>7.4</td>
<td>Checkpoint sizes (in MB) of the benchmarks (largest data size).</td>
<td>237</td>
</tr>
</tbody>
</table>
List of Tables

xx
1 Introduction

In the past decades, modern science and product development have undergone a major change [111]. Nowadays, most scientists and engineers rely on computer-based simulations of experiments or simulations of products in development. High Performance Computing (HPC) solutions provide a means for simulations that relieve the scientists and engineers of the need to create expensive experimental setups. Instead, the experiment is modeled in a simulation program and the outcome of the experiment is simulated on a (high performance) computer system. Crash test simulations and research on earthquakes are just two examples.

Because of economical reasons, car manufacturers simulate the crash behavior of cars from very early designs to the roll-out version of the car [24]. Without crash test simulations on high-end computing systems, the development would be slower and much more expensive. Being a destructive test, the test car cannot be used for more than a single run. Hence, for each test a new prototype has to be built and tested. In addition, if the designers change aspects of the prototype, old prototypes cannot be adapted and have to be rebuilt in most cases. Simulations help the manufacturers to save time and money.

Another example of an important application of computational science is the simulation of earthquakes [9]. With simulations, researchers gain insight into the processes that lead to an earthquake and try to obtain a better understanding of the impacts. Better knowledge about earthquakes helps to construct more reliable buildings and lives may be saved. In the future, it might even be possible to forecast earthquakes and to evacuate the affected areas.
With the advent of Grid computing [71], novel computing opportunities arise for researchers. Throughout the computational Grid, researchers gain access to a vast computing infrastructure with a variety of different systems. If the local computing resource is not capable of performing the desired computation, researchers can access a different system and execute their application on it. In the future, computational Grids are expected to deliver high-end computing power at the user’s finger tips.

Alas, the boundaries between the individual systems of a computational Grid are visible to users. Most systems are hidden behind a firewall or are located in a private network that is not fully accessible from the Internet. Hence, although the user has access to these systems, they do not yet form a transparent computing infrastructure in which a user can easily start an application. The different architectures such as Symmetric Multi-Processors (SMP), clusters, or constellations of a Grid have to be respected by the user, when choosing the right platform to execute an application. Heterogeneity is a major problem, since it limits the user’s capabilities to freely move between the accessible systems.

In addition to the problem of heterogeneity, a Grid user is faced with the job scheduling mechanism of the accessible systems. The job scheduler assigns the resources of a machine to applications (the so-called jobs). It expects a so-called job script that describes how an application has to be started on the system. For each job, the user has to specify the set of resources the application demands at runtime. Together with the job script, the user has to pass the number of Central Processing Units (CPU)—or nodes in case of a cluster—and the maximum runtime of the job. The job scheduler then exclusively reserves the requested node/CPU count for the specified time. If the application exceeds the reserved time it is prematurely terminated by the job scheduler. A terminated application loses all computed work that has been computed thus far.

It is difficult for a user to provide an exact estimation of an application’s runtime. Runtime depends on the application’s algorithms, the input data, the compiler options used to compile the application, and many more factors. Moreover, environmental factors influence the runtime in unpredictable ways. For example, the overall load on the cluster and the load on the interconnection network of the cluster nodes have an impact on the application’s performance and its runtime.
Introduction

Figure 1.1: Cumulative histogram: reserved versus used runtime.

Users solve the problem of premature termination by overestimating the runtime of the application. As an educated guess, the runtime is multiplied by a factor (e.g. two or three) to ensure that the reserved time is sufficient even if the application runs slower than expected. Some users even request the maximum permissible runtime that the job scheduler accepts.

An evaluation of the log files of the Woodcrest cluster at the Friedrich-Alexander-University of Erlangen-Nuremberg shows that only 19% of the reserved time is actually used by the application. The remaining 81% are used as a safety margin by the users. For example, if a user requests five hours of computing time for an application, the application is likely to terminate after one hour. Although the remainder of the reservation is released and may be used by other applications, overestimated runtimes have negative effects on the cluster’s schedule (see below).

Figure 1.1 shows a cumulative histogram that compares the requested runtime and the used runtime of the applications executed on Woodcrest. Although most users request more than thirteen hours for their applications, most applications finish in the first hour of the reservation. Figure 1.2 shows a histogram of the number of nodes requested by the users of the Woodcrest cluster. Most jobs request 9 to 16 nodes and 33 to 64 nodes. Long jobs with high demand of nodes cause negative effects on the cluster schedule and queue waiting times.
Figure 1.2: Distribution of the jobs sizes on the Woodcrest cluster.

Usually, the job scheduler of a cluster prefers short-running over long-running jobs. Short jobs that request only a few CPUs increase the utilization of the cluster. In contrast, long-running jobs with a high demand of CPUs cause unproductive reservation holes [215]. Due to the reservation holes, users have to accept long waiting times, since the scheduler has to hold back the job until enough CPUs become available for the requested time.

Figure 1.3 shows an example of a cluster schedule with a mixture of long-running jobs (A, B, and D) and two parallel jobs with a large number of CPUs (C and E). The dark gray part of each job indicates the time, the job actually needs to finish its computation. As can be seen, there are two reservation holes in the schedule (indicated by the shaded rectangular areas). While A and B are active, a set of nodes is idle until A has finished computation and C starts to execute.

A closer look reveals that job E could fit into the reservation hole, if the user had not reserved an overly high runtime for the job. With a migratable job, it would be possible to apply backfilling and to start E while C is waiting for execution. The term backfilling is used to indicate that a job that arrived lately is preferred for scheduling to fill a reservation hole [144]. If E takes too long to finish, E could be migrated to another reservation and finish computation there. The same applies to job D.
Job C could be scheduled to run earlier, if it requested a smaller number of nodes. Hence, if the job could change the requested number of nodes, the scheduler could adjust the degree of parallelism of a job to fit the current schedule. Although job B is already running, it could use more CPUs if no other job is scheduled to run in the reservation hole.

In this work, we propose a novel solution to the scheduling and resource estimation problem. The presented solution consists of two orthogonal parts. (1) Reparallelization alters the number of threads during the execution of a parallel region in OpenMP [154] applications on clusters. New nodes can be employed for computation as they become available. If nodes are needed for another application, the runtime system can scale-down the application to release nodes and to satisfy the request. (2) Checkpoint-based migration between clusters is supported. An application can stop execution on one cluster and resume operation on another cluster. Both the origin and the target of the migrations can be of different architectures, which is especially important in Grid environments.

Reparallelization and migration blend well with each other, since the number of nodes is likely to change when an application migrates from one system to another. With reparallelization, the application may request fewer or more nodes for the next reservation, depending on the cluster’s overall load and queuing times.
1.1 Terminology

Before we are able to investigate how an application may alter its degree of parallelism, we first define a set of terms that are important for the rest of this work. We closely follow the definitions given in [136]. However, we extend the definitions where needed to allow for a better differentiation of the concepts and techniques presented throughout this work.

**Task.** A task is the most generic granularity in parallel programs. From a software engineering point of view, a sequential algorithm needs to be split into a set of independent tasks that may be executed concurrently at runtime. An example is the divide-and-conquer version of the merge sort algorithm [72], as the operations on the different partitions may be considered independent tasks.

**Unit of Execution (UE).** The tasks of an application are assigned to different UEs for execution. An unit of execution may be a thread or a process, depending on the programming model used. This perspective is sufficient for most parallel applications. For this work, however, we further need to distinguish between logical and system units of execution. This gives rise to a more flexible way of handling parallelism that needs to be mapped onto an execution environment.

**Logical Unit of Execution (LUE).** The logical unit of execution is a UE that is visible at the level of the programming model and that the programmer can explicitly use to implement a parallel application. In multi-threaded programming models, an application-level thread (e.g. a Java thread) is considered an LUE. Another example is the use of OpenMP directives to specify a parallel region and to create a number of OpenMP threads. These threads are considered as the LUEs that execute the parallel region.

**System Unit of Execution (SUE).** For execution, the LUEs visible at the programming level have to be mapped to a system unit of execution to eventually execute the work contained in the LUE. Considering the thread programming model, the LUE corresponds to an application-level thread that is assigned to a kernel-level thread for execution. We define the notion of logical UEs and system UEs to be able to introduce an abstraction between the parallelism of an application and the parallelism of the execution environment (see Section 1.2).
1 Introduction

**Processing Element (PE).** A *processing element* executes the stream of instructions of an SUE. In most cases, the CPU as the main computing resource plays the role of the PE in a computing environment. An external instance (e.g. the operating system or a DSM environment) is responsible for assigning the SUEs to one or more PEs for execution. This assignment process is traditionally referred to as *scheduling* in the operating system literature [173, 192].

**Malleability.** The term *malleability* describes an application’s ability to dynamically reconfigure the number of executing SUEs while the application code is running. However, the term does not necessarily prescribe an alteration of the number of LUEs that have been spawned by the application. We will review malleability in more detail in the following section.

### 1.2 Classification of Parallel Applications

Feitelson and Rudolph provide a scheme [68] to classify parallel applications depending on their ability to execute with different numbers of UEs. It does not limit classification to either logical or system UEs, but provides a generic scheme to classify any parallel application. Figure 1.4 depicts the categories of this classification scheme.

**Rigid.** For a *rigid* application, the programmer specifies a fixed number of UEs for execution. This number is stringent, that is, the application does not accept any smaller number of UEs nor does it utilize any higher number of UEs. Rigid applications are implemented if the number of UEs is crucial for the correct functioning of the application. For example, an application might assume a quadratic number of UEs to process a quadratic data structure.

---

**Figure 1.4: Classification scheme for parallel applications.**

- **Parallel applications**
  - **Rigid** decided by user
  - **Evolving**
  - **Moldable** decided by system
  - **Malleable** decided by system

---
1.2 Classification of Parallel Applications

**Evolving.** Evolving applications may alter the number of used UEs. The change can happen either during execution or before the application starts, that is, for each run of the application a different number of UEs may be specified. However, the change is solely triggered by the application and the execution environment has to fulfill the new requirement or the application has to abort. Some programming models provide primitives to alter the number of UEs during the execution of the application. In OpenMP [154], the application may call `omp_set_num_threads` to change the number of UEs before the next parallel region is executed.

**Moldable.** Applications that are moldable adapt to any number of UEs to run on. The programmer does not specify the number of UEs, as it is determined by the computing environment when the application starts and remains fixed during execution. Most applications that use the MPI programming model [143] are moldable applications. OpenMP applications that do not explicitly specify a thread number by using `num_threads` or `omp_set_num_threads` can be considered moldable, as the runtime system decides at the application start-up on the number of threads to use during the execution.

**Malleable.** If an application is able to automatically adapt to the changes of available PEs in the computing environment and alter its number of UEs accordingly, it is called malleable. Sometimes this is also referred to as dynamic partitioning as changing the number of UEs often requires to alter the work and/or the data partitioning as well.

For rigid or evolvable applications, the programmer sets up the requirements of the application and the system has to respect these requirements. However, we can further sub-divide moldable and malleable applications into automatic and semi-automatic ones. In an automatic approach, the application’s runtime system is capable of adapting the application’s work/data distribution to the changed number of UEs without any additional programming effort. If the programmer has to respect a certain programming model or the application needs to provide callback functions to the runtime system, the system is said to be semi-automatic.

While this classification scheme categorizes applications by their ability to alter the degree of parallelism, there are different ways to eventually implement such applications. Figure 1.5 shows a further sub-categorization of
implementation approaches that make an application malleable. A description focusing on the technical details of malleable OpenMP applications is presented in Section 4.2.

**Inter-phase alteration.** *Inter-phase alteration* solely changes the number of UEs between phases of parallel execution, that is, in sequential code areas. For example, if the number of OpenMP threads is changed in an OpenMP code, the change is respected only where a new parallel region is entered; the currently executed region is not affected by the change.

**Multiplexing.** In the *multiplexing* case, LUEs are mapped to SUEs. If the number of processing elements changes, the LUEs are remapped to the changed number of PEs by modifying the scheduling policy. From an application point of view, the degree of parallelism remains unchanged, since the number of LUEs is considered rigid or moldable. Instead, the parallelism of the underlying execution model is adapted.

**Reparallelization.** In contrast to inter-phase alteration and multiplexing, *reparallellization* modifies the number of units of execution (e.g. threads), while the application is in a phase of parallel execution. It is important that the work and the data distribution is adapted to the new degree of parallelism. Otherwise, new LUEs remain idle and do not contribute to the execution of the parallel region.

**Hybrid.** A *hybrid* approach mixes one or more of the aforementioned approaches for malleability into a single compound approach. For example, multiplexing may be used to alter the number of SUEs and reparation is triggered to create new LUEs when the number of SUEs is about to reach the number of existing LUEs.
1.3 Contributions

In this work, we propose a solution to the problem of reserving computing resources by enabling OpenMP applications to reparallelize and migrate between reservations on the local system and remote systems of the Grid.

With our solution, the user can start the application at an arbitrary point of the Grid environment. From there on, the application automatically searches for free resources on accessible clusters of the Grid, transparently migrates to these systems, and automatically requests computing time. If the application is about to exceed the reserved time slice, it migrates to another system with free resources. This process is repeated until the application has finished computation. As each free resource potentially offers a different number of processing elements, the application has to adapt its degree of parallelism to the available number of processing elements to fully exploit a free resource.

The contribution of this work consists of the following orthogonal parts that together make an application migratable.

With reparallelization, OpenMP programs [154] can transparently alter the degree of parallelism during the execution of parallel code regions. Our compiler for OpenMP augments the compiled code with so-called adjustment points at which an application can alter its degree of parallelism while a parallel region is active. It is important that adding the reparallelization code to an application does not introduce a too high performance penalty. Our solution adds a negligible overhead of 3% on average to the application.

A checkpoint-based [110] migration technique stores the state of an application to disk and freely moves the state between clusters of different architectures. We solve the problem of creating a platform-independent checkpoint of the application state without introducing unacceptable overheads. During normal operation, an application’s performance must not be decreased noticeably. We solve this problem by compiler-generated checkpointing and uncheckpointing functions that convert the platform-dependent state into a generic portable format. Our checkpointing solution is almost free of overheads, the average overhead imposed on the application is about 1%.

A prototype implementation of a migration framework automates the process of application migration. The framework searches for free resources in the Grid and migrates applications to these if beneficial for the application.
We define a flexible architecture for the framework to enable research on migratable applications. We implement a default migration strategy that uses a simplistic performance model to assess the computing power of a cluster.

As a research vehicle, we use the Jackal DSM system [203] and implement an OpenMP compiler for it. We specify an OpenMP binding for the Java programming language [78] and target a specification that tightly integrates the OpenMP programming model and the Java programming philosophy. If necessary, we adapt OpenMP’s semantics to fit Java’s needs. We propose extensions to the existing set of OpenMP features to allow for a better use of Java’s object-oriented concepts in OpenMP programs.

With JaMP, we provide a state-of-the-art implementation of OpenMP/Java in the Jackal compiler and DSM system. JaMP’s compiler supports compiling OpenMP/Java to regular multi-threaded code. The generated code employs Java’s native Application Programming Interface (API) for multithreading to execute parallel regions of the program. In addition, to the thread binding, we propose a novel $m:n$ binding that decouples OpenMP threads from the underlying executor threads. It becomes possible to freely schedule the OpenMP threads on the set of executor threads. The implementation of the JaMP runtime system is kept flexible, that is, many aspects of the runtime system can be exchanged without the need to recompile the JaMP implementation. At the same time, we take special care to make the implementation as efficient as possible.

1.4 State of the Art

Since our contribution consists of different parts that interact, state-of-the-art approaches and implementations are discussed per module in the chapters of this work. In this section, we present a general overview of the approaches to alter the degree of parallelism of an application and to migrate it.

Application migration in Grid environments imposes a set of requirements. First, the programming model should not be different from non-migratable applications. The migration has to be transparent, that is, no change to the program code is needed. Furthermore, as each system of the Grid potentially is of a different architecture, migration between heterogeneous systems has to be possible. Finally, reparallelization has to provide a means to alter the degree of parallelism and to adapt it to the new computing environment.
Reparallelization and migration techniques for OpenMP programs are not yet available for Grid environments. Our work is the first that supports true reparallelization of active parallel regions. Except for inter-phase alteration (already supported by OpenMP), [191] targets irregular algorithms in OpenMP and proposes a solution to remove threads from a parallel region; adding threads to the execution is not supported.

Various projects support migration of applications between computing systems. We selected MOSIX [22], Sprite [57], Cactus [12], DGET [63], and P-GRADE [121] for discussion, as these projects are the state of the art.

MOSIX and Sprite migrate single-process applications between machines of the same cluster. Both only support homogeneous systems, as the application cannot be migrated between machines of different types; the architecture and the operating system version have to be the same. As the application might use threads internally, migration of OpenMP codes between machines is possible with MOSIX and Sprite.

Cactus and DGET migrate applications between systems and can alter the degree of parallelism during a migration. However, both violate our requirement that the programming model must not be changed. Both enforce their own programming paradigms, in which the application provides callbacks to application-specific functions that extend a prescribed framework.

P-GRADE supports migration of applications that use MPI [142] as their programming model. An application is checkpointed and transferred to the target system. At the target system, the application has to restart with the same number nodes it started with. Changing the degree of parallelism is not supported. This limits the applicability of this approach, as the application cannot exploit a higher node count on the target system. Conversely, the application might suffer from a performance penalty, if the target system offers fewer nodes and is overloaded by the application.

In contrast to existing approaches, we provide a novel solution that respects all of the above requirements. Without introducing unacceptable overheads, our reparallelization and migration approach checkpoints an OpenMP application and transfers its state to the target system. The application can resume from the checkpoint and freely adapt its degree of parallelism to the computing environment. Hence, the application optimally exploits computational power of the target system.
Java has become a wide-spread programming language during the past years. It benefits from its “compile once, run everywhere” approach, that enables programmers to execute the compiled code on any architecture for which a Java Virtual Machine (JVM) is available. Java provides programmers with a portable high-level programming environment. Java is often characterized as having a low execution performance. However, this lack of speed results from the Java implementation chosen, and it has been shown that Java is able to achieve reasonably good performance for scientific applications [36].

Java is not only suited for business-application development, but also makes it interesting for the implementation of HPC applications. With the upcoming multi-core and many-core architectures, there is a strong need to develop parallel Java programs. Java is explicitly designed to offer multithreading to programmers [147]. Alas, the programmer is forced to implement all aspects of threading, that is, defining the individual threads, assigning work, and dealing with synchronization primitives (i.e., `Object.wait()`, `Object.notify()`). This makes multi-threaded programming error-prone and difficult to handle.

This chapter defines an OpenMP specification for Java (`OpenMP/Java`). The Java language is extended in such a way that OpenMP-aware Java compilers can transform the sequential Java code into a parallel version. If necessary, OpenMP/Java extends some of the OpenMP directives to better fit the Java programming philosophy. OpenMP/Java respects the requirements of both the Java Language Specification (JLS) [78] and the OpenMP 2.5 specification [154]. OpenMP/Java conserves the semantics of most of the OpenMP constructs, directives, and clauses.
This chapter first highlights the state of the art of OpenMP and multi-threaded programming (Section 2.1) and provides a short introduction to the OpenMP programming model and the ideas behind it (Section 2.2). As the OpenMP specification lacks a binding for Java, we show what such a binding might look like (Section 2.3). We closely follow the OpenMP specification wherever possible. However, we identify some extensions for a much tighter integration of OpenMP into Java.

2.1 State of the Art

OpenMP is a programming model that extends a sequential base language (C, C++, and Fortran) with parallelization directives that an OpenMP-enabled compiler can use to emit a parallel program. Java is a programming language that already provides parallel programming with threads in its programming model.

Section 2.1.1 provides an overview over the programming models that are explicitly designed for parallelism. We focus on a selection of concurrent and parallel programming models that target multi-threaded programming. In Section 2.1.2, we address issues that have to be solved when two programming models already specified are merged with each other. We skip Java bindings of libraries such as MPI for Java [40, 206] or CORBA [149], since the base language is not modified.

2.1.1 Parallel Programming Models

Because of the vast extent of parallel programming models available, a serious in-depth discussion of parallel programming models is impossible in the scope of this work. Thus, we restrict ourselves to a short selection of well-known programming systems and point the reader to surveys on Concurrent Object-Oriented Languages (COOL) [29, 157] for in-depth discussions. OpenMP/Java (similar to OpenMP for other languages) cannot be considered a COOL, as OpenMP/Java is primarily intended for the parallelization of regular, loop-based programs.

OpenMP [154] offers a high-level, thread-oriented API. OpenMP extends a base language with compiler directives for semi-automatic parallelization
OpenMP for Java (see Section 2.2.2) of regular shaped loops and code blocks. The task of parallelization, that is, identifying opportunities for parallelism, is left in the hands of the programmer, while the compiler takes care of the technical details such as thread creation, data distribution, data access, synchronization, etc. The programmer is relieved from these tasks, which makes developing OpenMP applications less error-prone and therefore much easier. At the time of this writing, the base languages supported by OpenMP are C, C++, and Fortran. In this work, we provide a specification of a new binding of OpenMP for Java and focus on a tight integration of OpenMP into Java’s language philosophy.

Although a variety of OpenMP compilers for different architectures and languages exists (see Section 3.1), an OpenMP specification that targets Java as base language is missing. With JOMP [35, 148], a partial Java implementation of OpenMP exists. However, JOMP neither provides a full specification of OpenMP/Java nor cleanly integrates Java’s object-oriented programming philosophy into OpenMP/Java. The specification presented in this section uses the foundations laid by JOMP but achieves a tighter integration of both the Java and the OpenMP programming model.

Besides OpenMP there are other (low-level) multi-threaded programming models. A majority of parallel applications is written in a sequential language such as C or C++ and parallelized using a threading API. In the following, we shortly present the POSIX thread API [58, 97] and Java’s multi-threading support [78].

The POSIX thread API defines a set of threading primitives that programs can exploit to create and manage threads. The programmer has to implement a C function that contains the thread’s code. Calling the API functions, a function pointer is passed to a thread creation function and the thread is created in the program’s process space. It is the programmer’s task to provide code for assigning work to the threads, to synchronize the threads, and to avoid conflicts (e.g. race conditions). While offering the highest degree of flexibility to a programmer, such low-level APIs make multi-threaded programming error-prone and difficult to handle for programmers.

With the `java.lang.Thread` class, Java offers low-level multi-threading constructs to programmers as well [122, 147]. The programmer is responsible for correctly implementing the `run` method of the thread class (or implement the `Runnable` interface). Although being an object-oriented interface
to multi-threaded programming, the basic steps of implementing a parallel program remain the same as for the POSIX thread API.

The Intel Threading Building Blocks (TBB) [163] define both loop-parallel and task-parallel constructs for multi-threaded C++ programs. TBB offers C++ templates that parallelize for loops or create tasks for concurrent execution. The TBB parallelization templates force the programmer to hoist the parallel code out of the surrounding sequential code and to move it to C++ objects. Because of the invasive code transformation, creating a sequential program from the same sources as the parallel program is no longer possible. In contrast, a non-OpenMP compiler safely ignores OpenMP’s parallelization directives and emits a sequential program version without the need to change the program code.

Since the advent of Java 1.5 [138, 186], Java provides a new high-level API (located in the package java.util.concurrent [187]) for task parallelism. The programmer does no longer decompose the application into a set of threads, but instead identifies code fragments that can be executed concurrently. If moved to a so-called FutureTask object, these fragments can be handed over to an executor service for (concurrent) execution. If the result of the computation is needed, it is possible to block and wait for the result to be computed.

Using the POSIX thread API, the Java threading API, or the task-based API of java.util.concurrent is error-prone and cumbersome, as data and work distribution have to be handled manually by the programmer. In case of parallel applications operating on regular data structures such as arrays, matrices, or grids, the programmer is still forced to manually divide data and work among threads, while programming models such as OpenMP already provide a semi-automatic solution to this problem. Our OpenMP/Java binding alleviates this lack by providing the OpenMP programming model for Java.

Cilk for C [27] and JCilk for Java [49] offer a task-based programming model for their respective base language. C and Java are extended with Cilk keywords (e.g. spawn, sync). With spawn a procedure call is executed concurrently while the caller proceeds until a matching sync is encountered. On a conceptual level, the tasks of the java.util.concurrent package and JCilk offer roughly the same programming model. The major difference is that java.util.concurrent tasks are implemented by a Java API supported by
any Java compiler, whereas JCilk is a language extension that needs a special
compiler accepting the new keywords.

To summarize, OpenMP offers the following advantages to a programmer: Instead of having to deal with the low-level details of concurrent or parallel
programming, OpenMP offers a high-level declarative syntax to inform a
compiler about how to parallelize (parts of) a program. By adding the par-
allelization hints in a piecewise fashion, the programmer can incrementally
parallelize a sequential program. This is impossible with the traditional
thread APIs, as they mostly require a major redesign of the code. Pro-
gramming models such as the task API of Java, TBB, or JCilk still require
a major change to the program code, as the concurrently executable code
fragments have to be moved to newly created classes (Java tasks, TBB) or
functions (JCilk). Java gains the advantages of OpenMP with the integra-
tion of OpenMP/Java as defined in this chapter.

2.1.2 Language Extensions for Java

If a programming language such as Java is extended with new features, it
must be guaranteed that the new features are compatible with the language
features of the base language. In case of the OpenMP/Java binding, the
programming model of OpenMP has to be incorporated into the existing
programming philosophy of Java.

With respect to a clean and correct integration of the OpenMP programming
model into the Java programming model, the following major aspects have
to be addressed:

• As OpenMP is defined for C/C++ and Fortran, C++ is the only
object-oriented programming language supported by OpenMP. How-
ever, Java’s notion of object-orientation is different from the one used
by C++. Hence, an OpenMP integration has to respect Java’s object-
oriented philosophy. This also holds for other programming paradigms
for which a Java binding should be specified.

• Java heavily uses exceptions, as virtually all byte-code instructions
may either cause an exception or a runtime error. Hence, a language
integration has to explicitly define how to react on exceptions that cross
the boundaries of the programming models. As OpenMP completely
ignores exceptions, OpenMP/Java has to define how to react, when an
exception is not caught inside a parallel region, but is passed to the surrounding sequential code.

- Memory consistency (see Section 2.2.3) is an issue, since parallel programming language have to define when modifications of data become visible to other units of execution (i.e., threads or processes). Extending Java, the memory model of the added language has to be compatible with that of Java or has to incorporate Java’s memory model.

- Java is already defined as a programming language that offers support for multi-threading at the language level. An integration of another parallel programming paradigm has to respect Java’s notion of threads and provide a means to cleanly interact with Java threads.

Various other projects have augmented the Java programming model with constructs of other parallel programming models. In the following, a selection of these projects is covered to explore the design space of integrations of programming models.

JavaParty [158] extends the core language of Java with the `remote` keyword to mark a class as remotely allocatable (i.e., on a different cluster node). JavaParty breaks with compatibility to plain Java and needs special compilers that understand the new keywords. JavaParty extends Java’s object handling by allocating objects on other machines and providing transparent access through proxy objects. JavaParty passes exceptions that occur on a remote machine through the proxy objects back to the executing thread and relies on Java’s mechanisms to handle the exception. The memory model and threading model of Java are adopted, as JavaParty is explicitly designed for extending Java.

HPJava [41] is a language that introduces an SPMD-style programming model to Java. HPJava extends the core Java language with new keywords for data distribution and work distribution. HPJava is also designed as an extension to Java. Hence, the HPJava paradigms are developed to fit Java’s existing specification.

JCilk also extends Java with a new programming paradigm. Although based on the Cilk language for C, JCilk is designed to be compatible with Java’s programming philosophy. Except for the new language constructs, JCilk does not define how JCilk tasks and Java threads interact, it relies on Java’s
memory model, and does not prescribe any notion of data sharing. However, JCilk explicitly provides for the semantics of exception handling for exceptions that arise in concurrent JCilk tasks and that have to be passed to the sequential code area [49]. If exceptions are raised by concurrently executed tasks, JCilk arbitrarily selects one of them for handling in the sequential code. OpenMP/Java adopts this behavior to define clean semantics for handling exceptions that occur in parallel regions.

Whereas the aforementioned languages are explicitly designed with a Java binding in mind, our OpenMP binding for Java has to consider two distinct already existing programming models at the same time. Each feature of OpenMP and Java has to be incorporated for a smooth integration of both paradigms. OpenMP/Java, as proposed in this chapter, is the first successful attempt to provide a complete binding of OpenMP for Java that respects all aspects of Java and OpenMP at the same time. In addition, we extend the expressiveness of OpenMP’s constructs to better suite Java’s object-oriented programming philosophy.

2.2 Overview of OpenMP

This section gives a short introduction to the OpenMP programming model. It is intended to be neither a complete reference of OpenMP nor an introduction to OpenMP-based programming.

After a short historical excursion (Section 2.2.1), we introduce the programming model of OpenMP (Section 2.2.2). The current section is closed by investigating OpenMP’s memory model (Section 2.2.3).

2.2.1 History of OpenMP

At the time the specification of OpenMP started, all vendors of parallel computers provided their own parallel programming models. As a matter of fact, each programming model was incompatible with the others. Hence, development of portable parallel applications was limited, because parallel code running on one machine could not be compiled on another machine. It was a major development task for programmers to port such applications to fit the programming model of other machines from a different vendor.
2.2 Overview of OpenMP

Figure 2.1: Evolution of the OpenMP specifications over time.

To alleviate the lack of compatibility, a first attempt was made to standardize parallel programming on SMPs by proposing the ANSI X3H5 draft [46]. Some vendors later joined to propose a new programming model that was expected to be available on a variety of machines and architectures [196]. The consortium used the results from the X3H5 draft and released version 1.0 [152] of the OpenMP specification in 1997 (see Figure 2.1). In this version, a specification was available for Fortran only, as Fortran was considered to be the standard HPC programming language at that time. The specification already contained most of the currently available parallelization directives, the library routines, and other compiler extensions such as conditional compilation. However, it did not specify a memory model (see Section 2.2.3). One year later, a specification of OpenMP for C was made available to the developers [153]. It mainly provided C-style versions of the directives and library routines. Instead of free-form source comments, the C specification relied on the pragma concept of C/C++ compilers.

The OpenMP 2.5 specification [154] did not introduce new OpenMP features but merged the Fortran and C/C++ specifications into a single document. It contained a first attempt to define the memory model of OpenMP programs. However, this definition was given informally in natural English and was not fully formalized. However, a formalized specification is desired for tools that automatically check for the correctness of OpenMP applications.

The new OpenMP 3.0 specification [155] provides new parallel constructs such as the task construct [18, 172]. A similar construct is supported by the Intel compiler suite [99] in a restricted form. The taskq directive of the Intel compiler and the accompanying directives and clauses will provide a means to parallelize programs that, for example, use iterators or other types of queue-like data structures that cannot be parallelized with work-sharing constructs. OpenMP 3.0 specifies a new OpenMP clause to fuse nested
class SumArrayExample {
    int sumArray(int[] array) {
        int sum = 0;
        //omp parallel for shared(array)
        //& reduction(+:sum)
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }
}

Figure 2.2: Example of a parallel reduction of array elements.

Figure 2.3: Example distribution of a loop of length 20.

loops into a single parallel loop to better exploit the potentially reduced loop overhead and the higher parallelism of the combined loop.

2.2.2 Programming Model

With OpenMP, the programmer adds parallelization annotations to a sequential program. It is the task of the compiler to transform the code such that a program is created that runs in parallel. Although OpenMP does not define how this transformation is to be implemented, most compilers use a native threading library (e.g. the POSIX threading library [58, 97]) for creating and managing OpenMP threads. As the OpenMP specification proposed in this work is based on OpenMP 2.5, we only introduce features that are already available in OpenMP 2.5.

Figure 2.2 shows a simple program to sum up the elements of an array. As it can be seen from the figure, the programmer parallelizes the sequential loop
2.2 Overview of OpenMP

by adding the **parallel for** directive in line 4. The compiler then partitions
the loop in such a way that each worker thread receives the same amount of
work. Figure 2.3 shows an example of how the computation for an array of
length 20 is distributed among four workers. The **reduction** clause in line 5
instructs the compiler to sum up the intermediate results of each worker.

The OpenMP programming model can be considered a variant of the well-
known fork/join programming model [136, 161]. Figure 2.4 shows how the
code of Figure 2.2 is executed with eight threads at runtime. The sequential
parts of the program are executed by a single thread (the so-called **master**
thread) until a parallel region is reached. At the beginning of the parallel
code location, the master thread spawns a set of **slave** threads and distributes
the work among them. Subsequently, the master and the slaves begin execu-
tion of the parallel code fragment. OpenMP requires a barrier at the end of
the parallel execution. After each thread has reached the barrier, the slaves
terminate and the master thread continues with the next sequential code
region that follows the parallel region until it encounters the next OpenMP
parallel construct.

Besides parallelization constructs, the OpenMP specification also defines
a set of supporting constructs and directives that allow for more compli-
cated parallel execution paths. For example, a programmer can specify
**single/master** to implement code fragments that shall be executed sequen-
tially within a parallel region whereas the rest of the region is executed in parallel. With barrier additional barriers can be added within a parallel region and critical provides a means to implement mutual exclusion for a code block in a parallel region.

The OpenMP specification also defines a set of so-called runtime library routines, that is, functions that are contained in the support library shipped with every OpenMP-compliant compiler suite. The library routines are grouped into three categories: execution environment routines, lock routines, and timing routines. The first group provides a means to query and modify the execution environment. For example, a call to `omp_get_thread_num` delivers the ID of the thread that executes the call and `omp_get_num_threads` gives the number of threads currently executing the program region. The lock routines implement a more sophisticated locking than the critical construct can provide (e.g. nested locks). Timing functions measure the wall clock time in the execution environment and are included in the timing routines group to provide a portable interface.

### 2.2.3 Memory Consistency Model

The OpenMP 2.5 specification defines a parallel programming model on top of purely sequential languages. C/C++ and Fortran are sequential languages in that they do not offer any means to directly express parallelism with language constructs. Parallel programming is only possible by means of external libraries that are not tightly integrated into the language definition. This gives freedom in terms of how to implement parallelism, but also requires the library to define parallel semantics.

A Memory Consistency Model (MCM) determines the order in which memory references issued by a parallel program can be executed by the underlying system [211]. In other words, the memory model specifies at which time data is written back to memory and updated values become available for reading. This is very important especially for concurrent programming models, as the memory model defines the moment when a value that was updated by one thread is visible to another. There are several different consistency models that provide different semantics with respect to the program behavior [3, 141]. The most strict memory model is Sequential Consistency (SC), which guarantees that any interleaved execution order of threads will compute the same result as a potential sequential execution. As an example of a
2.2 Overview of OpenMP

![Figure 2.5: OpenMP memory model: shared memory and thread private memory.](image)

OpenMP defines two different types of memory: shared and thread-private (see Figure 2.5). Shared memory is visible among all OpenMP threads and can be concurrently read and written. The thread-private part of the memory is not shared among the threads. Hence, reads from and writes to this type of memory are not observable by other threads.

The MCM for shared memory defines at which time an update to data is visible to other threads. As OpenMP’s MCM is a relaxed memory model, a thread’s view on the shared memory can differ from the current state of

relaxed MCM, Weak Consistency (WC) softens the strictness of SC in that it does not ensure instantaneous memory updates, but requires the updates to only become visible upon encountering a synchronization point.

C/C++ and Fortran do not offer any consistency model for parallel programming. Hence, the OpenMP specification provides a detailed description of its memory model [154]. As C/C++ and Fortran do not define a memory model, the OpenMP specification is free to define a memory model without the need to deal with an existing MCM of the base language. The OpenMP MCM is based on weak ordering [3]. As its description is given informally, it gives room for ambiguities and misinterpretations. However, some effort has been made recently to provide a formal definition as well [30]; the authors work out a set of ambiguities that limit the use of the informal definition of OpenMP’s memory model. With respect to our overview, we also provide an informal description of the memory model in this section. Please refer to [30] for a complete discussion of this topic.
the shared memory. Threads are permitted to cache parts of the shared memory in a private, thread-local memory area (e.g. CPU registers). Only at so-called memory fences (or memory barriers) a thread is required to make its view of the memory consistent with the shared view again. At that point, the thread has to flush, that is, write back the cached data to shared memory. Additionally, it has to invalidate data that is cached for reading.

Memory fences are established by the flush directive that ensures that the executing thread updates its view of the memory. Barriers (created by the barrier directive), explicit locking (i.e., critical), and entry into a construct (e.g. entering a master construct) form an implicit memory fence as well. In addition, some calls to runtime routines, such as omp_set_lock also imply a memory fence.

### 2.3 Adapting OpenMP to Java

At the time of this writing, a full specification of OpenMP is available for C, C++, and Fortran only. Although the JOMP project [35, 148] attempted to implement OpenMP for Java, it did not provide a complete specification. In the remainder of this section, we propose a complete Java binding of the OpenMP API. It is based on the OpenMP C++ binding, since C and C++ are closer to Java than Fortran is.

Let us first discuss the requirements to tightly integrate OpenMP into Java (Section 2.3). We show how the OpenMP pragma syntax (Section 2.3.2), the various OpenMP constructs (Section 2.3.3), and the OpenMP runtime library (Section 2.3.4) can be adapted to the Java language. We then investigate how to determine the number of threads for execution (Section 2.3.5), present changes and extensions made by OpenMP/Java (Section 2.3.6), and define clean exception handling semantics (Section 2.3.7). Memory model issues (Section 2.3.8) and the integration of Java threads and OpenMP threads (Section 2.3.9) are discussed. We close this section by summarizing missing features and limitations of our specification (Section 2.3.10).

Some of the ideas of this section have already been presented in [117]. A prototype implementation of OpenMP/Java in the Jackal DSM system is summarized in [116]. Chapter 3 presents the prototype implementation (called JaMP) in more detail.
2.3.1 Design Objectives

The most important requirement for the OpenMP/Java specification is that an OpenMP/Java program must also be a valid Java program. That is, if the OpenMP/Java program is compiled with a regular, non-OpenMP Java compiler, a purely sequential program should be compiled without errors. This requirement is set up by the existing OpenMP bindings and has to be adopted by OpenMP/Java as well. Hence, only features of OpenMP that are compatible with the JLS [78] can be included into OpenMP/Java. Otherwise an OpenMP/Java program cannot be considered a regular Java program anymore.

Another requirement is set by the programmers that have to use OpenMP in the Java environment. OpenMP/Java has to be syntactically and semantically close to the other OpenMP bindings. An experienced OpenMP programmer should feel familiar with the OpenMP/Java binding. A high learning curve needs to be avoided. Thus, all constructs and directives follow the same semantics as their OpenMP counterparts. However, some of the OpenMP features may not be available in OpenMP/Java due to compatibility or performance reasons (see Section 2.3.10). Other constructs may have to be extended for a better integration into Java.

From a Java programmer’s point of view, OpenMP/Java should naturally extend the Java programming philosophy. Language features that are prescribed by the JLS must be reflected by the OpenMP/Java specification in a way that a Java programmer feels familiar with their respective integration into OpenMP.

2.3.2 OpenMP/Java Pragma Syntax

The OpenMP specification for C/C++ uses pragmas as the major language construct. Normally, pragmas can be used to instruct the C/C++ compiler about how to process the source code. For example, in some compilers they can be employed to turn off compiler warnings or they can provide information for more aggressive optimizations. An OpenMP-compliant compiler uses pragmas to inform the compiler on how to transform the sequential program into a parallel one. In Fortran, OpenMP uses free-source form directives, that is, special line comments that are treated the same way as pragmas [154]. In addition, OpenMP defines a set of macros that can be
used to conditionally include codes when compiling the source code with an OpenMP compiler.

As a key requirement of OpenMP/Java is full compatibility to the JLS, any OpenMP/Java program has to be a valid Java program and has to follow the well-known syntax of the existing OpenMP bindings (see Section 2.3.1). The JLS does not define any means to express pragmas or conditional compilation in Java programs. In addition, Java’s language philosophy discourages using an external preprocessor such as the C preprocessor to implement the missing features. Such preprocessors rearrange the source code in such a way that the programmer is not able to map compiler errors to the source. Hence, an equivalent replacement that is as expressive as the pragma-based syntax has to be found.

There are three possible solutions to implement the OpenMP pragma concept in Java: language extensions, Java annotations, and line comments. We will investigate their applicability in the next paragraphs.

A language extension modifies a base language to include additional features that are not available in the base language. Examples of extensions to Java to provide concurrent programming features are the JavaParty language [158] or JCilk [49]. A major drawback of this approach is that it breaks compatibility, that is, a standard Java compiler is not able to compile a JavaParty source file anymore. As compatibility is one of our major concerns, language extensions are not viable for OpenMP/Java.

From version 1.5 [138, 186] onwards, Java specifies a syntax for annotations that attach user-defined metadata to the source code. These metadata can be used by the compiler or the runtime system to gather additional information about the code or to apply code transformations during compilation. For example, the @Deprecated annotation tells the compiler to issue a warning that a given class is deprecated. Another popular example is the usage for the development of Enterprise Java Beans (EJB). The EJB 3.0 specification [188] intends to apply annotations, for example, to describe the relations between different beans (e.g. @OneToOne, @ManyToMany, etc.). Annotations cannot be used for OpenMP/Java because of two reasons. First, annotations can only be added at the class scope or at a method’s definition; a single statement or code block cannot be annotated. Second, each annotation has to be reflected by an annotation class that is found in the Java class path. Although an OpenMP-compliant compiler could ship the required annotation classes, a standard Java compiler would have to provide
a dummy implementation of OpenMP annotations. These limitations render annotations unusable for an OpenMP implementation.

The only viable solution is to define a special kind of line comment that is treated as an OpenMP parallelization hint. Using comments as pragmas fulfills all of the above requirements. As one can freely choose the syntax for the comment, one can use OpenMP’s set of keywords. Compatibility is also ensured, because a regular Java compiler safely ignores the comment while an OpenMP/Java compiler analyzes the comment and creates a parallel version of the program.

With respect to the OpenMP/C++ syntax, OpenMP/Java uses comments that start with //omp to identify parallelization hints. If a line comment starts with //& it is treated as the continuation of the OpenMP/Java line comment that is before the //&. With this feature, an OpenMP/Java parallelization hint can span multiple lines. A similar approach is used by the OpenMP binding for Fortran and JOMP’s [35, 148] implementation.

A special form of the aforementioned line comments is also used to implement the conditional compilation feature for OpenMP/Java programs. The OpenMP specification defines a set of macros that are activated when the program is compiled with an OpenMP-enabled compiler. The macros can be used to include or exclude code fragments. To simulate this feature with line comments, OpenMP/Java treats comments that start with //# as conditional compilation guards. Every statement that begins with //# is only considered by an OpenMP/Java-enabled compiler. Regular Java compilers, again, safely ignore such statements and do not emit code for them.

2.3.3 Constructs and Directives

By design, our OpenMP/Java supports most of the constructs and directives that are available in the current version of the C/C++ binding of OpenMP. The syntax of the constructs and directives remains unchanged in OpenMP/Java. The only exception from this rule is the different start token for the OpenMP pragmas.

The central construct in OpenMP/Java is the parallel construct that provides a means to execute a sequence of code in parallel. It supports all types of data-scoping clauses that are also available in the C/C++ binding of OpenMP. For variables marked as shared, the same variable is seen
by all threads. For private variables, every thread receives an initialized private copy of the variable. Although the OpenMP specification only requires an uninitialized copy, for a Java programmer it is more natural to have an initialized variable instead, as this is also the default behavior for instance variables and class variables. To privatize a variable and make its current value available in the parallel region, the variable can be marked as firstprivate. As the implicit this variable of a Java method is used frequently, it is always treated as a shared variable, and is automatically passed to the parallel region without the need to include it in a shared clause.

OpenMP/Java solely supports the for work-sharing construct known from the OpenMP C/C++ binding. The for construct provides a means to express the loop-parallelism pattern [136] in the OpenMP programming model. The workshare construct of the OpenMP/Fortran binding is not supported by both OpenMP/C++ and OpenMP/Java, since this construct is special to Fortran’s array syntax that cannot be expressed naturally in C++ and Java. The for construct is used to distribute the work of a Java for loop among the workers of a parallel region. As required by OpenMP, only a single loop counter is allowed in the initialization expression. The termination condition and the increment expression have to be invariant, that is, the shape and size of the loop’s iteration space [21] have to be known before the loop is executed. The loop counter is automatically privatized as enforced by the OpenMP specification.

The for construct supports different clauses that assign the iteration space to threads in a different way. The distribution can be influenced by means of the scheduling clause. With static scheduling, the iteration space is evenly distributed among the workers, that is, each thread receives a contiguous chunk of the same size. If the programmer specifies a chunk size for scheduling, the iteration space is cut into chunks of that size and the resulting chunks are assigned to the threads in a round-robin fashion. The dynamic schedule partitions the iteration space into chunks of an equal size specified in the clause and conceptually stores the chunks in a central work pile. Each thread then requests a chunk from the work pile, computes the results, and grabs the next chunk as soon as it is finished with the current one. A guided schedule also assigns chunks dynamically to threads but the chunk size depends on the number of unprocessed chunks and the number of threads. Finally, the runtime schedule enables a runtime decision on how the loop should be distributed. In this case, the value of the environment variable OMP_SCHEDULE is expected to contain the scheduling type.
At the end of parallel regions, there are implicit barriers at which all the threads that execute the parallel region wait for each other. For work-sharing constructs, the implicit barrier can be avoided by adding the `nowait` clause to the construct. However, `nowait` only effects barrier placement if the work-sharing construct is not a compound construct such as `parallel for`. Compound constructs always involve a barrier synchronization.

With the `reduction` clause, OpenMP/Java combines partial results that have been computed by the individual workers that execute a parallel region. OpenMP/Java supports all arithmetic reductions defined by the OpenMP binding for C/C++. The order in which the reductions are applied to the private intermediate results is undefined according to the OpenMP semantics [154] and depends on the actual reduction algorithm used by the underlying OpenMP/Java runtime.

OpenMP/Java also supports `parallel sections`. Whereas with the constructs `parallel` and `for` each thread executes the same sequence of code in parallel, with `sections`, the programmer specifies a set of `section` constructs, each of which is executed by a potentially different thread. To define code regions that are only executed by a single thread, `single` and `master` constructs can be used. Contrary to `master`, `single` permits the specification of clauses for data-scoping and has an implicit barrier at the end of the region’s code.

User-defined barriers are defined by means of the `barrier` directive. At the location of the directive, each thread waits until all other threads also reach the barrier. The `critical` construct can be used to execute a sequence of code with mutual exclusion, that is, the code sequence is only executed by one thread at a time.

### 2.3.4 Runtime Support Library

Beyond constructs and directives, an OpenMP implementation also has to provide a runtime support library. To provide a programmer with all the features of the OpenMP programming model, the OpenMP API has to be brought to Java as well. Porting the OpenMP API to Java is governed by the design objects presented in Section 2.3.1.

The runtime support library of the OpenMP C/C++ specification can be directly adopted by OpenMP/Java. However, Java does not support globally
scoped functions as C/C++ does. Hence, the only solution viable is to implement the runtime library as a set of static methods in an OpenMP/Java-specific library class. In OpenMP/Java this class is called `omp.Omp` (see Figure 2.6). The constructor of the class is private to disallow creation of instances of the class. An OpenMP/Java compiler automatically adds static import statements for the OpenMP/Java routines. Thus, the routines are automatically available without the need of prefixing `omp.Omp` for each invocation. To better integrate the functions into the Java programming philosophy, the names follow the naming conventions of the Java style guide [185] with a starting `omp` to avoid name clashes with other visible functions.

Figure 2.6 lists parts of the supported runtime library routines that are supported by OpenMP/Java. For brevity, the bodies of the individual methods and the public static access modifiers have been omitted from the code in Figure 2.6. As can be seen, all routines required by OpenMP are supported. Since Java's philosophy is to indicate errors by throwing exceptions instead of passing back an error code through the function's return value, the methods throw a runtime exception of type `OmpException` if an error has occurred during the call. Using a runtime exception avoids the need to enclose each method call with a try-catch block which might seem unnatural to an OpenMP programmer.

The only extension to the set of runtime support routines is for object allocation, which might be one of the major performance bottlenecks in a Java application. Some Java implementations use a centralized heap for all object allocations. Hence, to avoid race conditions, all allocations on the heap of the JVM have to be performed in a mutual exclusive way, that is, the heap has to be protected by a lock. In the worst case, a multi-threaded program runs virtually sequential, because of lock contention [39]. Although JVMs with thread-local allocation pools already exist [55, 124], programmers and OpenMP/Java compilers cannot generally rely on such JVMs.

In C/C++, it is possible for the programmer to change the object allocator by overwriting the `new` and `delete` operators or using in-place allocation [100]. This effectively solves the problem of allocation contention. In Java, however, this is impossible, as Java does not support overwriting the `new` operator. To alleviate this problem, OpenMP/Java provides a per-thread object pool that resembles the ideas of thread-local heaps and can be used to allocate objects in a sort of thread-local storage. With the function `ompObjectPoolGet`, an object is requested from the pool. If
2.3 Adapting OpenMP to Java

```java
package omp;

public final class Omp {

    private Omp() {}

    // runtime support functions
    int ompGetThreadNum();
    int ompGetNumThreads();

    // ...

    // Lock support functions
    void ompInitLock(OmpLock lock);
    void ompDestroyLock(OmpLock lock);
    void ompSetLock(OmpLock lock);
    void ompUnsetLock(OmpLock lock);

    // ...

    // object pooling functions
    Object ompObjectPoolGet(Class cls);
    Object ompObjectPoolGet(Class cls, int elts);
    void ompObjectPoolPut(Object obj);

    // support functions (custom cloning)
    void ompAddCustomCloner(Class cls,
                                CustomCloner clnr);
    CustomCloner ompGetCustomCloner(Class cls);
    void ompRemoveCustomCloner(Class cls);

    // reparallelization support functions
    void setAdjustStrategy(AdjustStrategyFactory factory);
    AdjustStrategyFactory getAdjustStrategy();

}
```

Figure 2.6: Java runtime support functions and additional functions for object pooling.
ompObjectPoolGet is called with argument elts set, an array of size elts elements is allocated from the pool. An object is returned to the pool by calling ompObjectPoolPut on the object’s reference. Synchronization only occurs if an object that was allocated from a thread is released by another.

### 2.3.5 Determining the Number of OpenMP Threads

Each OpenMP implementation has to determine the number of threads executing a parallel region. The OpenMP specification comments on how the number of threads is determined at runtime (see Figure 2.7). We will now shortly describe the flow chart.

When encountering a parallel region, the OpenMP runtime checks for the presence of the `if` clause. If the clause is present and evaluates to `false`, the parallel region is executed sequentially. The clause is used for choosing between sequential and parallel execution depending on the program’s state (e.g. on the amount of data that has to be processed in the parallel region).

The OpenMP runtime then has to check whether or not nested parallelism should be considered. To determine the number of threads, the OpenMP runtime uses a set of so-called *Internal Control Variables* (ICV). If a parallel region is encountered while another parallel region is currently active, it depends on the ICV `nest-var` if the encountering thread is either sequentially executed or executed in parallel. The `nest-var` variable can be modified by either setting an environment variable `OMP_NESTED` or by calling the runtime function `ompSetNested`. The OpenMP specification does not require the implementation to handle nested parallelism; it is possible to ignore nested parallel regions and execute such regions sequentially with only the encountering thread.

The OpenMP runtime also checks the ICV `dyn-var`, which enables the implementation to use a varying number of threads during the execution of the program. The variable can be set by the environment variable `OMP_DYNAMIC` or by calling `omp_set_dynamic`. If the environment variable is set to `false`, the programmer-specified `num_threads` clause determines the number of executing threads. If no `num_threads` clause is present, the implementation chooses the thread number depending on the ICV `nthreads-var`, which can be set by either an environment variable (`OMP_NUM_THREADS`) or by calling `ompSetNumThreads`. If `dyn-var` is set to `true`, the implementation may
Figure 2.7: Determining the number of threads for a parallel region (from [154]).
choose any thread number between one and the value of `num_threads` when entering the parallel region.

Basically, OpenMP/Java adheres to the flow chart of Figure 2.7 to determine the number of threads for a parallel region. OpenMP/Java’s runtime provides `ompSetNested`, `ompSetDynamic`, and `ompSetNumThreads` functions. However, as Java already provides properties as a feature, we use the property feature instead of using environment variables. This provides a better integration into the Java programming philosophy while keeping the basic syntax. For example, the number of threads can be set from outside the virtual machine by adding `-DOMP_NUM_THREADS=n` to the command line arguments of the virtual machine at program start-up.

### 2.3.6 Changes and Extensions to OpenMP/Java Constructs

While the preceding sections have shown that the OpenMP syntax can be ported to Java without syntactic and semantic changes, this section investigates extensions to the OpenMP/Java constructs for a tighter integration into the Java programming model. Although Java and C++ are object-oriented languages, Java’s concepts differ from C++ in many details. Hence, these differences need a special handling when defining OpenMP for Java.

Our OpenMP/Java specification makes changes and extends parts of the original OpenMP 2.5 specification for C++. OpenMP/Java offers slightly different data-scoping rules and extends them to better handle Java objects. Parallel iterations provide a means to iterate over Java collections in parallel. Finally, OpenMP/Java extends reductions to object-oriented reductions.

#### Data-scoping

The first change affects the semantics of OpenMP data-scoping clauses, that is, the semantics of `shared`, `private`, etc. To recapitulate, `shared` is used to make the storage location of a variable visible to all threads of a parallel region. To make a private copy of a variable and initialize it with its value from outside the parallel region, `firstprivate` can be used. In an object-oriented language, `shared` and `firstprivate` have to operate on objects as well. Therefore, in the C++ binding of OpenMP, the behavior of `firstprivate` is as follows: The value of a `firstprivate` pointer `x` is copied
and passed to the parallel region as a value. For \( x \) being of an *object type*, its copy constructor is invoked to create a copy of \( x \) and the copy is passed to the parallel region. The same argument holds true for the semantics of the \texttt{lastprivate} clause, which passes values from the parallel region to the code after the region.

Using the C++ \texttt{firstprivate} semantics for OpenMP/Java would imply that only the reference is privatized whereas the underlying object is still shared among the individual worker threads. Since Java only knows about \textit{references to objects}, as a consequence, Java programmers would not have any means to privatize an object through the \texttt{firstprivate} clause. Obviously, this is undesirable, as this would imply a major restriction of the programming model and limit its use. Thus, the semantics of \texttt{firstprivate} have to be adapted to allow for privatization of Java objects. Instead of just making a copy of the \textit{reference} to an object, \texttt{firstprivate} creates a copy of the \textit{object} behind the reference.

To follow the new semantics of \texttt{shared}, \texttt{firstprivate}, and \texttt{lastprivate}, OpenMP/Java defines the following behavior for the data-scoping clauses:

- For a \texttt{shared} object reference, the reference is copied and passed to the parallel region.
- The \texttt{firstprivate} and \texttt{lastprivate} clauses always create copies of objects by calling the \texttt{Object.clone()} method of objects. To use the \texttt{clone} method, the objects’ classes have to implement the \texttt{Cloneable} interface.

The \texttt{Cloneable} interface might not be implemented by all classes and it might not be desirable or even possible (e.g. for classes contained in third-party libraries) to add the implementation of the interface to the class definition. For such classes, the programmer can implement a specialized \texttt{CustomCloner} to enable cloning (see Figure 2.8(a)). In addition, the custom cloner may be used to change the default copying behavior to create a deep clone instead of a shallow one. The example in Figure 2.8(b) shows the use of a custom cloner to pass \texttt{firstprivate} instances of \texttt{Integer} objects to a parallel region. The custom cloner is activated for the variable in the \texttt{firstprivate} clause (line 15) by prefixing the variables’ identifiers with a Java expression separated by a colon (see reduction clauses). Any Java expression (e.g. method calls, creation of new objects) that evaluates to a value of type \texttt{CustomCloner} is allowed at this place. For each thread,
package omp;

interface CustomCloner<T> {
    T clone(T other);
}

(a) Interface for creating custom copies for a type T.

import omp.CustomCloner;

class IntegerCloner implements CustomCloner<Integer> {
    public Integer clone(Integer other) {
        return new Integer(other.intValue());
    }
}

class CloningExample {
    void example() {
        IntegerCloner cloner = new IntegerCloner();
        Integer i = new Integer(1);
        Integer j = new Integer(2);

        // #omp parallel firstprivate(cloner:i,j)
        {
            int sum = (i.intValue() + j.intValue());
            System.out.println("Sum is "+ sum);
        }
    }
}

(b) Example for custom object copying.

Figure 2.8: Interface definition for custom object copying and example of usage.
2.3 Adapting OpenMP to Java

the custom cloner’s `clone()` method is then called to eventually create the thread’s private copies of the objects. The same syntax is also valid in the lastprivate clause.

In addition to explicitly specifying custom cloners at the directive level, custom cloners can be declared at the class level. This simplifies writing data-scoping clauses by reducing redundant cloner specifications, since programmers do not need to add the cloner’s reference to every single data-scoping clause that references a particular class. Instead, the programmer can use special runtime library routines (see Figure 2.6) to register a custom cloner with a class type (see `ompAddCustomCloner`). Whenever a firstprivate or lastprivate clause mentions an object of a class type with an attached custom cloner, the custom cloner’s `clone` method is automatically invoked by the OpenMP/Java runtime system. To unregister a cloner, `ompRemoveCustomCloner` can be used. Only one instance of a cloner per class type may be associated; registering another instance automatically overrides the former registration. Associating two cloners for the same class would result in ambiguous calls to the `clone` method of the custom cloner.

Parallel iterators

As most Java programs make heavy use of Java’s collection API and for-each loops that have been introduced in Java 1.5, it is desirable to provide a means to parallelize for-each loops over collections as well. For example, a molecular dynamics application [208] might exploit this API to store collections of molecule objects instead of using flat arrays of doubles. Without extending OpenMP/Java to support the Java collection API such programs cannot be parallelized with the existing work-sharing constructs, as OpenMP currently does not define any construct that parallelizes such loops.

To tightly integrate the collection API and for-each loops into the Java OpenMP binding, we extend the set of constructs with `#omp iterator` and `#omp parallel iterator` (similar to `#omp for` and `#omp parallel for`, see Figure 2.9). When using these constructs, the iterator, which is purely sequential in regular Java, is replaced by a parallel version of the for-each loop. The compiler generates code that conceptually converts the collection into an array by calling the `toArray` method of the collection. The for-each loop is rewritten into a regular `for` loop with an ordinary loop counter instead of the iterator. The resulting loop structure is then
parallelized by the for work-sharing construct (or parallel for in presence of parallel iterator). For the iterator constructs the same clauses are valid as for the for work-sharing constructs.

The current version of the Intel compiler suite contains a more general taskq construct [18, 172]. The taskq construct is intended to enclose a parallel region that contains a set of task environments. While executing the enclosing structural block, the encountering thread appends an entry to a centralized task queue. The worker threads dequeue task by task and process the code contained therein. If the enclosing taskq contains a loop, the encountering thread executes the loop sequentially.

The more generic construct is needed for C and C++, as both lack a language-specific iterator interface for loops over collections. For example, in most C++ programs, iterators are implemented by classes that overload the ++ and == operators. Java, however, contains in its API a standardized Iterator interface and provides a syntax for for-each loops. Hence, the iterator constructs offers sufficient expressiveness and naturally integrates for-each loops into OpenMP by directly extending concepts that are already known to both OpenMP and Java programmers.
2.3 Adapting OpenMP to Java

The OpenMP 3.0 specification [155] also contains support for task-based programming. In contrast to the Intel taskq extension, OpenMP 3.0 only defines a single task construct that creates and submits a task to a centralized task queue. The OpenMP threads request a task from the list and execute the code associated with it. As task is more generic than Intel’s taskq version, a future extension of OpenMP/Java should consider providing the task construct in addition to the iterator construct.

Object-oriented reductions

Reductions are used to compute a final result out of per-thread intermediate results by combining the intermediate results with associative and commutative operations. For example, when computing the scalar product of two vectors in parallel, each thread can compute a partial sum of its loop chunk into a private variable. When the work-sharing construct terminates, the OpenMP runtime reduces the intermediate results to the final result. So far, OpenMP only defines reductions for primitive data types and built-in operators such as addition and multiplication (the Fortran binding also allows minimum and maximum computations).

However, in the Java programming philosophy it is most natural to decompose the application into a set of interacting objects. Hence, most of the applications’ data is expressed using objects and not as primitive data types. As reductions currently cannot be applied to object types, current practice forces programmers to explicitly implement reductions. User-implemented reductions are error-prone and often inefficient, because the runtime system cannot change it to the most efficient reduction algorithm. For example, the runtime might use a linear reduction algorithm for small numbers of threads while it uses a tree-based algorithm for larger numbers of threads. Changing the reduction algorithm is possible because only associative and commutative operations are valid in reductions.

The custom reduction interface depicted in Figure 2.10(a) extends the current notion of OpenMP reductions by adding support for reductions on object types. Programmers can implement the Reducer interface to extend the set of built-in reduction operators with additional user-defined operators that are specific to a particular application’s data structures. User-defined reduction operators can be used to implement reductions for arbitrarily complex data types or to provide additional types of reductions (e.g. minimum
or maximum reductions). The interface offers a method `reduce` that takes two objects as input and computes a reduction of these objects. The `init` method is used to initialize the value of intermediate results with the neutral element for the reduction within the parallel region.

Following the idea of separation of concerns, the approach of external reduction operators offers three major advantages:

- The definition of existing classes (e.g. classes from third-parties or the Java API) do not have to be changed or extended to make reductions available for them. Hence, even classes for which the source code is not accessible can be used as input for reductions by simply implementing a specialized reducer for them.

- An approach that implemented reductions as a `reduce` method within the class, would limit reduction capabilities to just one kind of reductions, because in Java it is not possible to implement more than one `reduce` method without changing signatures. Changing signatures, however, inhibits the OpenMP runtime to actually call the additional methods, as they are not known upfront when the OpenMP runtime system is compiled. With the interface, it becomes possible to implement different kinds of reductions for the same class of objects just by implementing additional reducers.

- The OpenMP runtime can, for performance reasons, select the best reduction algorithm without the need to provide such functionality in the application.

Figure 2.10(b) gives an example of a reduction for the `BigInteger` class from the `java.math` package. Lines 3–12 implement a user-defined reduction operator. It provides a `reduce` method that adds two `BigInteger` objects and returns the result. The `init` method is used to supply the neutral element for the addition, that is, a `BigInteger` representing the value zero. Every time the OpenMP/Java runtime reduces intermediate results of threads, it first creates the neutral element with `init`. Then the intermediate results are reduced to compute the final result. Please note, that the number of invocations of `reduce` and `init` depends on the internal reduction algorithm. The same holds for the relative order of the individual invocations.
package omp;

interface Reducer<T> {
    T reduce(T a, T b);
    T init();
}

(a) Interface for object-oriented reductions.

import omp.Reducer;

class BigIntegerReducer implements Reducer<BigInteger> {
    public BigInteger reduce(BigInteger a, BigInteger b) {
        return a.add(b);
    }
    public BigInteger init() {
        return new BigInteger("0");
    }
}

class ObjectReductionExample {
    void example() {
        BigInteger value = ...;
        BigIntegerReducer reducer = new BigIntegerReducer();
        //#omp parallel reduction(reducer:value)
        {
            ...
        }
    }
}

(b) Example of object-oriented reductions.

Figure 2.10: Interface definition for user-defined reduction operators and usage example.
2.3.7 Exception Handling

The current OpenMP specification does not define how to react on exceptional events within parallel regions. While C and Fortran do not offer language constructs to throw or catch exceptions, C++ exceptions are only infrequently used by APIs and are not expected to be thrown within parallel regions, as this breaks OpenMP’s requirement that a parallel region must not be terminated prematurely. This does not only include exceptions but also jumps (e.g. goto or break) from the inside of a parallel region to the outside as well as terminating the function by executing a return in the parallel region.

Java, however, makes heavy use of exceptions. Exceptions are not only visible at the programming level as part of the method signature, but are also used to indicate system events and errors (e.g. NullPointerException, ArrayIndexOutOfBoundsException, etc.). Such system-level events may occur at many different byte-code instructions. To offer clean semantics on such exceptional events, an OpenMP implementation for Java therefore has to explicitly deal with the presence of exceptions and has to take special care of exceptions that are thrown by a parallel region and are caught in the surrounding code.

Unfortunately, it is not immediately clear how to react on an exceptional event within a parallel region. Current programming languages cannot handle more than one exception at the same time. Hence, it is not obvious to define semantics for concurrent exceptions thrown in a parallel region. This is confirmed by literature that shows that providing clean semantics for languages with exceptions that potentially occur in concurrent code paths is not a trivial problem [101, 193, 212]. First, terminating the exception-throwing threads by stack unwinding [50, 150] causes a deadlock of the other team members, as they infinitely wait for the terminated threads at the next barrier. Second, it is not permitted to proceed to the next barrier, because the cause for the exception might still be affecting the program and, thus, continuing the program without handling the exception is not an option. This results in an undefined behavior, which is also not tolerable. The desired behavior is that the whole team should abort as soon as possible if one member has encountered an exceptional event. For performance reasons, however, it is not acceptable that each thread continuously polls for excep-
2.3 Adapting OpenMP to Java

Exceptions that might have occurred in other threads [67]. For this purpose, we define the semantics of exception handling in OpenMP/Java as follows:

- If one thread throws an exception, it catches the exception and sets a cancellation flag for all other threads and registers the exception at the thread team. If the thread is not the master thread, it terminates.

- All (other) threads check for the occurrence of an exception at so-called cancellation points. Cancellation points are the start and the end of `//#omp barrier`, `//#omp critical`, `//#omp parallel`, and work-sharing clauses.

- When a thread reaches a cancellation point and notices that another thread has requested cancellation, the thread terminates by throwing an internal exception. This ensures that the thread’s stack frames are unwound and execution of the thread is terminated.

- The registered exception is re-thrown by the master thread after all threads have exited the parallel region. This ensures that the parallel region is cleanly exited and the exception is safely announced at the surrounding sequential code. If more than one exception was indicated in the parallel region, the master thread arbitrarily selects an exception and re-throws it, again causing stack unwinding and execution of a matching handler in the sequential code.

The JCilk [49] parallel programming language defines similar semantics for handling exceptions that occur in parallel code. Although JCilk does not define the notion of cancellation points, it asynchronously terminates computation in case of an exceptional event. Such computations are aborted as soon as possible after the exception has occurred. If more than one exception is thrown, JCilk also arbitrarily selects one to be handled by the corresponding catch block.

2.3.8 OpenMP/Java Memory Model

As already mentioned (see Section 2.2.3), C/C++ and Fortran are sequential languages without a memory consistency model. Hence, OpenMP can freely define an MCM and enforce it on top of the base language. For Java, the situation is different, as Java is explicitly designed for parallel programming. The JLS [78] contains a specification of Java’s MCM, the so-called Java...
Memory Model (JMM), that every Java implementation has to adhere. If OpenMP has to be ported to the Java language, both consistency models have to be compatible to each other. Otherwise either one has to be dropped, which either conflicts with the JLS or with the OpenMP specification.

From the MCM point of view, it is possible to integrate OpenMP into Java without dropping any requirement of one of the MCMs, since the JMM and the OpenMP memory model [30] are conceptually very close. Similar to the OpenMP memory model, the JMM knows a global shared memory area (the heap), in which all Java objects are created and reside during their life cycle. This part of the memory is subject to the garbage collector [108] to free objects that are no longer used. Each thread also owns a thread-local memory area. Upon access to an object, the thread caches the accessed data therein for efficient access in the future. To synchronize the thread-local cache with shared memory, from time to time, each thread flushes the cache at specific points during its life-time (so-called synchronization points).

Important synchronization points of a particular thread are the entry and the exit of synchronized blocks and methods. The JLS also requires that starting a thread and terminating a thread are considered as synchronization points (see [78] for a complete list of synchronization points). On the level of Java byte code, synchronized blocks and methods are translated into monitorenter to acquire the monitor of an object and monitorexit to release it. During execution of monitorentry, the JVM flushes the thread’s modified data and discards all values read thus far. Upon monitorexit, only modified data is written back to the shared memory. Hence, after both operations, the thread’s view of the shared memory is consistent again.

OpenMP requires that a thread flushes its cache, when it encounters a flush or a barrier directive. As Java conceptually specifies the same behavior as OpenMP when a synchronization statement is entered or left, it is natural to reuse Java’s flushing facilities for OpenMP/Java’s flush operations. In naive implementations it is sufficient to use an empty synchronized block—such as synchronized(this) {}—for any OpenMP flush operation. As the JVM has to flush the executing thread’s private memory, all requirements of the OpenMP MCM are fulfilled. Unfortunately, Java does not offer byte-code instructions to explicitly flush the thread’s cache when needed. The only way to flush the cache is a synchronized block to force the compiler to emit monitorenter and monitorexit instructions to the byte-code stream. The lock operations associated with monitor entry and exit cause overheads that
2.3 Adapting OpenMP to Java

could be avoided if the JVM offered flush operations. In a native compiler environment, the compiler is able to exploit machine instructions for cache flushes and thus may achieve a higher performance.

2.3.9 Interaction between Parallel Regions and Java Threads

As an extension to purely sequential languages, OpenMP for C/C++ and Fortran does not deal with the integration of native threads and OpenMP threads. Java, however, contains language support for programming with multiple threads. Hence, an OpenMP/Java implementation has to explicitly define how Java threads and OpenMP threads may safely interact with each other. This is especially true, since many Java applications need to use threads in their implementation. For example, the GUI framework Swing [130] uses a Java thread for GUI event handling. If GUI events have to be triggered from within a parallel region, one of the OpenMP threads will have to interact with the GUI thread.

Hence, OpenMP/Java has to state which functionality of the Java threads is available to the programmer and what is disallowed. OpenMP/Java must define sufficient (but not too severe) restrictions on the usage of Java’s threading features so that no harmful interactions between OpenMP/Java threads and Java threads may result. The following is a list of restrictions on code within parallel regions for this purpose. Outside parallel regions, that is, in regular Java code, all of Java’s features may of course still be used as usual, as no OpenMP/Java threads are active in these code locations.

- The programmer cannot use `Thread.currentThread()` to differentiate between regular Java threads and OpenMP/Java threads that execute a parallel region. OpenMP/Java threads only own a logical thread ID (see Section 2.3.4, Figure 2.6), but may not be distinguished by their underlying thread object. This is important if multiple OpenMP/Java threads are multiplexed on a single Java thread (see Section 3.5).

Most uses of `Thread.currentThread()` simulate some form of thread-local storage by using the reference to the thread as a key into a hash table to retrieve a thread-local value. With OpenMP/Java, this is not necessary, since OpenMP data-sharing primitives can be used to provide the same functionality in parallel regions.
• The use of `synchronized()` is disallowed in parallel regions. Instead the various OpenMP/Java thread synchronization constructs must be used. If multiple OpenMP/Java threads are multiplexed on internal Java threads, `synchronized()` will not behave as desired and may cause deadlocks.

• The programmer may not use `Object.wait()` as it may cause deadlocks of the OpenMP/Java threads that execute the parallel region. As OpenMP/Java offers synchronization primitives, a programmer does not need to use `Object.wait()` in a parallel region.

• However, `Object.notify()` may be used as it is guaranteed to not block. Hence, it is possible for an OpenMP/Java thread of a parallel region to notify threads that have been spawned by non-OpenMP parts of the Java application (e.g. the event dispatcher of Swing).

Because of the dynamic code loading features of Java [128], a OpenMP/Java compiler is not able to provide the programmer with error messages if the above rules are violated in dynamically loaded code. It is, however, possible for a OpenMP/Java compiler to analyze the currently compiled parallel region for enclosed statements that violate the rules.

2.3.10 Missing OpenMP Features and Limitations

OpenMP/Java does not fully implement the whole OpenMP 2.5 specification; in this section we list the limitations of the current OpenMP/Java version. However, the limitations do not restrict the overall expressiveness of OpenMP/Java. The missing constructs and directives can easily be expressed by other constructs and directives or can be replaced by a short sequence of regular Java code. Hence, we have chosen to remove this features from the specification, to achieve a smaller and easier to understand OpenMP/Java specification.

OpenMP/Java currently does not accept declarations of orphaned regions. An orphaned region is an OpenMP region (e.g. a `for` region) that is not enclosed by a lexically surrounding `parallel` construct. In such cases, OpenMP-compliant implementations are forced to dynamically determine whether or not the region was reached from inside an active parallel region. If there is none, the orphaned region is executed sequentially without creating a thread team for parallel execution.
### 2.3 Adapting OpenMP to Java

Table 2.1: OpenMP constructs and directives supported by OpenMP/Java.

<table>
<thead>
<tr>
<th>Construct</th>
<th>OpenMP 2.5</th>
<th>OpenMP/Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>for</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallel for</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>workshare</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>parallel workshare</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>sections</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>section</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallel sections</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>iterator</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>parallel iterator</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>single</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>master</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>barrier</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>atomic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>flush</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ordered</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>threadprivate</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

We also have limited the functionality of OpenMP/Java on some issues to keep the complexity of the prototype implementation acceptable. The current implementation cannot privatize instance fields or class fields. In particular, `threadprivate`, `copyin`, and `copyout` clauses are not supported by OpenMP/Java. In addition, the `ordered` directive and clause are not implemented in the compiler.

OpenMP/Java does not support the features of OpenMP 3.0. It can be extended to support the new features such as pragmas for task-parallel execution and new runtime support functions, by adding the corresponding definition. There is no substantial change with OpenMP 3.0 that would break compatibility the Java programming model.

To summarize, Table 2.1 and Table 2.2 compare the support for OpenMP constructs, directives, and clauses of the OpenMP 2.5 specification with OpenMP/Java. The tables indicate whether the construct, directive, or clause is supported (“✓”) or is not supported (“✗”). The tag “✛” in Table 2.2
Table 2.2: OpenMP clauses supported by OpenMP/Java.

<table>
<thead>
<tr>
<th>Construct</th>
<th>OpenMP 2.5</th>
<th>OpenMP/Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>shared</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>firstprivate</td>
<td>✓</td>
<td>✛</td>
</tr>
<tr>
<td>lastprivate</td>
<td>✓</td>
<td>✛</td>
</tr>
<tr>
<td>reduction</td>
<td>✓</td>
<td>✛</td>
</tr>
<tr>
<td>if</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>copyin</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>copyout</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>threadprivate</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ordered</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

indicates an OpenMP/Java extension to the semantics of the OpenMP clause by adding features for a better integration into Java’s language philosophy.

2.4 Summary

We have presented a first successful attempt to integrate OpenMP’s and Java’s programming paradigms into the OpenMP/Java binding. We have shown that it is worthwhile to augment Java with OpenMP’s high-level programming model for multi-threaded applications. OpenMP/Java’s declarative syntax offers constructs and directives that hide Java’s APIs for multi-threading and that avoid an error-prone manual implementation of data and work distribution or thread synchronization.

As both Java and OpenMP offer different programming model specifications, an extension of plain Java with the OpenMP parallelization hints is only possible if the OpenMP features can be brought to the Java language without major changes to either of the languages. This is ensured by the three fundamental requirements we have defined in this chapter.

First, OpenMP/Java programs have to behave like regular Java programs when compiled with a non-OpenMP Java compiler. This requirement is motivated by the fact that OpenMP for C/C++ (and Fortran) is defined
as a set of compiler pragmas that are ignored by non-OpenMP compilers. Keeping this behavior of OpenMP compilers for the Java binding as well, is the basis for a semantically equivalent integration of OpenMP into Java.

Second, as a natural extension of the core Java language, a Java programmer should feel comfortable with the OpenMP programming model. An integration has to keep Java’s programming semantics where possible. The JMM has to be respected and Java-specific constructs such as for loops over collections must be supported. Another important requirement is a clean handling of exceptions that arise in parallel regions.

Third, OpenMP/Java has to closely follow the syntax and semantics of the OpenMP binding for C/C++. Experienced OpenMP programmers should be able to use OpenMP/Java without a too steep learning curve. The OpenMP/Java constructs and directives should keep the syntax of their OpenMP C/C++ counterparts. Finally, the semantics of the OpenMP programming model should be kept as it is where possible.

Although a former attempt to adopt OpenMP for Java (i.e., JOMP [35, 148]), partially solved problems of integrating OpenMP and Java, this chapter provided the first full specification for OpenMP/Java. While JOMP offers Java versions of most OpenMP constructs and directives, data-scoping and exception handling are not cleanly handled. Neither does JOMP deal with memory model issues. In contrast, we consider each aspect of such an integration of OpenMP and Java, and propose a clean and semantically complete specification that covers all important OpenMP constructs and directives. We also analyze the OpenMP memory model and the JMM and argue about the compatibility between the two memory models.

We have also extended the existing set of OpenMP constructs with new features to better integrate the Java’s philosophy of programming into the OpenMP programming model. We have introduced the new iterator construct and a compound iterator construct parallel iterator to parallelize for-each loops over Java collections. Additionally, we have also adapted the behavior of the data-sharing clauses to better fit Java’s notion of object-oriented programming. With CustomCloner and Reducer it is possible to fine-tune the behavior of data-scoping clauses and reductions for object types. Finally, we have proposed clean semantics for handling exceptions that arise from the execution of parallel regions.
3 Implementation of OpenMP/Java in the Jackal Compiler

In the previous chapter it has been shown that it is worthwhile to integrate the Java programming model and the OpenMP programming model. As it is crucial to understand how reparallelization of OpenMP parallel regions works, we will now take a closer look on how an OpenMP compiler can be implemented. In particular, this section discusses an implementation of OpenMP/Java for the Jackal DSM for Java [203]. The described OpenMP/Java implementation is called \textit{JaMP}. An OpenMP/Java implementation in the Eclipse Java Development Tools [61] for the Eclipse Platform [62] is presented in [56]. The Eclipse OpenMP/Java compiler follows the same design principles that are the foundation of \textit{JaMP}.

It is the task of the \textit{JaMP} compiler to transform OpenMP/Java code into multi-threaded code that can be executed on all machines that are supported by the Jackal compiler framework (e.g. IA-32, IA-64, EM64T, PowerPC). Jackal ensures portability across several architectures and different modes of compilation (DSM or shared memory systems). In addition, the mapping has to be efficient, that is, compilation of OpenMP constructs shall not introduce high overheads into the compiled program. Finally, our implementation follows state-of-the-art techniques that are used to implement OpenMP compilers in general. \textit{JaMP} can be used as a research vehicle on malleable OpenMP programs and the reparallelization techniques (see Chapter 4) can easily be generalized to other OpenMP bindings.

First, we describe how OpenMP compilers are implemented in general and how we implement \textit{JaMP} in the Jackal framework. We re-use the \textit{JaMP} implementation for the reparallelization features that alter the data and
work distribution while a parallel region is executed (see Chapter 4). Second, we propose an $m:n$ binding that uses multiplexing to assign OpenMP/Java threads to native threads for execution. The $m:n$ mapping adds another degree of freedom to execute the work load of an OpenMP parallel region. JaMP uses the multiplexing technique features to change the application’s degree of parallelism by adding and removing native threads as needed. However, it turns out that the Jackal DSM does not efficiently support the $m:n$ mapping (see Chapter 7). A multi-CPU shared-memory system could provide a more efficient execution environment for this mapping. We leave this question open for further research.

We first evaluate the state of the art in OpenMP compilation for DSM systems (Section 3.1) and give an introduction into the Jackal compiler and its runtime system (Section 3.2). Afterwards, we describe the general architecture of an OpenMP compiler and the design considerations for the implementation of JaMP (Section 3.3). Next, we discuss the traditional mapping of OpenMP to (Java) threads (Section 3.4) and then propose an $m:n$ mapping of OpenMP threads to Java threads (Section 3.5). Section 3.6 closes with an overview of the implementation of the JaMP runtime system.

### 3.1 State of the Art

Traditionally, OpenMP implementations target multi-processor systems with shared memory architectures. Implementations of OpenMP for cluster environments have to simulate the shared memory programming model on top of the distributed memory architecture of the cluster. This problem can either be solved by means of a compilation scheme that internally uses a communication library (called “direct compilation”) or by a middleware layer (the DSM system) that transparently handles remote memory accesses. Figure 3.1 shows a scheme to classify OpenMP implementations by their primary compilation target (Section 3.1.1). Section 3.1.2 discusses the reason for using Jackal’s DSM system as compilation target.

#### 3.1.1 OpenMP Implementations

Our system mainly targets efficient execution of OpenMP on distributed memory architectures. Obviously, an efficient mapping of OpenMP-parallel
regions to the underlying execution environment is the foundation to achieve a good application performance. Our compiler provides a 1:1 mapping of OpenMP threads to native threads as well as an \(m:n\) binding with multiplexing support (see Section 3.4 and Section 3.5).

An efficient data distribution scheme is crucial to application performance on distributed machines such as clusters. If compiling directly to a distributed memory system, the compiler is responsible for placing the communication primitives such that data distribution and data access are handled optimally. JaMP relies on the Jackal DSM to find an application-specific data distribution scheme.

Table 3.1 lists several OpenMP projects and their corresponding compilation targets; the last column indicates the support for compiling OpenMP/Java. We will now assess the selected OpenMP implementations, summarize the compilation techniques used, and compare them to our prototype implementation of JaMP.

Many commercial compilers (such as the Intel compiler [99] suite or the PGI compilers of the Portland Group [194]) can compile OpenMP C/C++ and Fortran codes, but are limited to shared memory architectures and cannot compile OpenMP/Java codes. JaMP primarily compiles for a cluster and is a full-fledged OpenMP/Java compiler.

Besides the aforementioned commercial products, there are several open-source implementations of OpenMP. GCC 4.2 [73] is a native open-source compiler that translates OpenMP code to machine code. OdinMP/CCp for C/C++ [33], the Omni OpenMP compiler for C/C++ and Fortran [171],
## 3.1 State of the Art

Table 3.1: Classification of OpenMP implementations.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Shared memory</th>
<th>Direct compilation</th>
<th>DSM</th>
<th>OpenMP/Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel suite</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>PGI suite</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>GCC 4.2</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OdinMP/CCp</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Omni</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OpenUH</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>JOMP</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OPMi</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[23]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[96]</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Intel CLOMP</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Omni/SCASH</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>[131]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>ParADE</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>JaMP</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Legend:
- ✓ supported
- x unsupported
and OpenUH [127] are also open-source OpenMP compilers. These are source-to-source compilers that preprocess the OpenMP source code and emit a transformed source program in the base language that makes use of a native threading API. The transformed program is then compiled into an executable program with a standard compiler. In case of OdinMP the source is parallelized using POSIX threads, whereas OmniMP allows the use of different threading APIs. The mentioned OpenMP compilers neither compile OpenMP/Java code nor can they emit an OpenMP program that can be executed on a cluster.

JOMP [35, 148] is the only OpenMP implementation (except ours) supporting OpenMP/Java-style programming, but is limited to compilation for shared memory systems. JOMP also is a source-to-source compiler emitting a preprocessed Java program in which the OpenMP/Java-like code is mapped to regular multi-threaded Java code. Due to technical details related to the preprocessing step, the programmer is forced to declare exceptions that might be thrown inside the parallel region and that then reach the sequential code. JaMP does not force programmers to declare exceptions and natively compiles code for execution on a cluster.

JaMP not only implements a traditional OpenMP thread binding, but also implements an $m:n$ mapping of OpenMP threads to native threads. No other compiler offers a comparable feature that can be used to implement malleable OpenMP applications. OMPi [85] is the only other project that targets a similar compilation pattern. Whereas JaMP supports changing the number of native threads during the execution of parallel regions, OMPi keeps this number fixed throughout a parallel region.

Although none of the OpenMP implementations mentioned above (except ours) is capable of emitting a compiled program that runs on a cluster system, approaches that compile OpenMP programs for clusters exist. The approaches follow different implementation ideas that are presented.

Using the idea of partially replicated data, the OpenMP compiler of [23] uses direct compilation to MPI [142] to hide the distributed memory nature of a cluster. An unmodified OpenMP program is analyzed for its work-sharing constructs and shared data. The compiler then compiles the program such that MPI is used to transfer data between the cluster nodes as needed. The OpenMP source-to-source compiler of [96] translates OpenMP programs into programs that exploit Global Arrays [146] for hiding a cluster’s distributed memory architecture. The OpenMP constructs and clauses are transformed
3.1 State of the Art

into calls to the functions provided by the Global Array API. While these projects need compiler analysis to handle communication and data distribution, JaMP relies on the Jackal DSM to handle these issues automatically.

Examples of OpenMP implementations that target distributed shared memory systems are Omni/SCASH [151], Intel’s Cluster OpenMP (CLOMP) [91], and the OpenMP implementation of [131]. Omni/SCASH targets the SCASH DSM as shared memory middleware layer. Intel’s Cluster OpenMP is based on the standard Intel compiler suite and uses a modified version of the TreadMarks DSM system [113] to provide the shared memory environment. In [131] the TreadMarks DSM is used as well. Finally, the ParADE S-DSM [112] explicitly targets execution of OpenMP codes on clusters.

Although the above mentioned projects compile OpenMP applications for clusters, the programmer has to modify the OpenMP codes to allocate global variables and memory in the DSM space. Omni/SCASH, CLOMP, and [131] define directives and runtime functions for this purpose. If shared data are needed, the programmer has to use the API functions to allocate the data in the DSM storage area. ParADE avoids this issue by making global variables private by default. If this is not desired by the programmer, shared clauses have to be added to give a hint to the compiler that a variable should not be private. Our system is more generic, as an OpenMP/Java program can run completely unmodified on the Jackal DSM.

3.1.2 Distributed Shared Memory Systems

Using a middleware layer (the S-DSM) that simulates a global address space on top of the distributed memory of a cluster is a well-known technique. There are several projects that target different aspects of such middleware layers. DSMs do not directly provide OpenMP functionality, but they can be used as targets for OpenMP compilers.

In general, S-DSMs cannot use specialized hardware for the object access checks, as they are expected to support commodity clusters. Many S-DSM systems use memory pages as the granularity of data transfer and memory consistency. The Memory Management Unit (MMU) of the Operating System (OS) is exploited to protect memory pages against reading or writing. If the program accesses a restricted page, the OS triggers the S-DSM’s runtime system by sending a signal and notifies it about the faulty access. After the
DSM system has handled the page fault by retrieving the page, the memory access is re-issued and the application continues.

SCASH [87], TreadMarks, and CAS-DSM [134] are examples of page-based S-DSM systems. TreadMarks and SCASH rely on the OS to receive notifications about accesses to remote data that is not cached on the local node. CAS-DSM is comparable to Jackal (see below), since the compiled source code is augmented with function calls to check for the availability of data.

In object-oriented environments (as targeted by OpenMP/Java), page-based DSM systems suffer from false sharing, as disjoint accesses to different objects in the same page potentially cause the page to be accessed by multiple nodes. False sharing causes the DSM protocol activity to increase and application performance is decreased. Therefore, we consider page-based DSMs unusable as a target platform for our OpenMP/Java implementation for compute clusters.

JavaParty [158] uses standard virtual machines to simulate an S-DSM environment. Java is extended with the `remote` keyword to tag classes for remote allocation. For objects created from these classes, the JavaParty compiler generates proxy objects [74] that handle the remote access of `remote` classes. For communication, JavaParty uses the standard Java RMI interface [183]. JavaParty cannot be considered a real DSM system, as it does not maintain a globally shared address space. Instead, remote object accesses are made transparent by RMI calls to an object’s storage node.

JESSICA2 [214] is a Distributed Java Virtual Machine (DJVM) running on the distributed memory of a cluster. JESSICA2 uses specialized JVMs that together form a DJVM and provide a so-called global object space for object allocation. Objects are identified by globally unique object references that are similar to Jackal’s global object references (see Section 5.7.1), but are not as flexible in the size of the cluster used.

Jackal [203] is an S-DSM optimized for the use as a compilation target for the Java programming language on clusters. Instead of memory-pages, Jackal uses Java objects as the granularity of data transfers. False sharing is reduced as each object is accessed on its own without affecting other objects. Section 3.2 provides a summary of the Jackal DSM. A detailed description of Jackal’s DSM implementation is given in Section 5.7.1. As Jackal offers both a compiler framework for Java compilers and a DSM runtime system for clusters, Jackal is the optimal solution for integrating JaMP.
3.2 The Jackal Compiler and Runtime System

Jackal’s compiler is a modularized, optimizing multi-pass compiler that supports compilation of Java source code to various architectures (e.g. IA-32, IA-64, EM64T) and different compilation modes (for DSM or shared memory systems). Figure 3.2 shows how a Java source file is processed by the compiler. The lexer and the parser [82] read the source code from the input file and create an Abstract Syntax Tree (AST) that corresponds to the program’s structure. The AST is then translated into high-level LASM, Jackal’s intermediate code representation. The back-end processes the LASM code, applies several optimizations to it, and continuously approaches low-level LASM. After all code-modifying back-end passes have been executed, the low-level LASM code is transformed to assembly code and passed to an external assembler and linker to create the final executable program.

One of the key features of LASM is its ability to express different levels of intermediate code in a single intermediate representation. At the highest level, LASM still knows about control structures such as loops, if statements, etc. During the optimization in the back-end, this high-level LASM code is continuously converted to lower levels at which the control structures are mapped to (un)conditional jumps and expression trees [8]. This level of intermediate code is subject to many ahead-of-time optimizations. After machine-independent optimizations have been applied, the intermediate code is mapped to the lowest level. At its lowest level, LASM is comparable to assembly code and machine-specific optimizations (e.g. register allocation, instruction scheduling) are applied. This is also the final stage of LASM before it is directly mapped to the external assembler’s syntax.

The compiler back-end supports a variety of program optimization techniques. Amongst them are language independent optimizations such as loop unrolling, invariant code motion, common sub-expression elimination, and others. In addition, the back-end also includes optimizations that are specific to object-oriented languages such as Java. For example, if the compiler is able to prove by means of escape analysis that an object reference is never visible outside the call stack below the current function, it allocates the object on the stack instead of the heap. This may save an expensive garbage collection after the object is used for the last time. Another example of a Java-specific optimization is bounds-checks removal that avoids unneces-
Figure 3.2: The Jackal compiler pipeline.
sary index checks when accessing an array. See [202] for a complete list of optimization techniques of the Jackal back-end.

The Jackal compiler and runtime system together implement a shared memory for Java programs on top of the distributed memory of a compute cluster. In order to provide the global address space, the compiler inserts access checks in front of each object access when translating Java code to LASM code. The access checks test whether or not the currently accessed object is already available at the local node. If not, the DSM runtime system is triggered to fetch the object from its home node, that is, the node that stores the master copy of the object. The runtime system also contains the complete Java API plus some additional APIs for the handling of the Jackal DSM. For example, it is possible to query general data about the current application (e.g. the number of executing nodes, process IDs, etc.), performance data (e.g. the number of executed access checks), or to issue prefetch requests on a set of Java objects [114]. The DSM runtime also provides an API for the programmer to influence object allocation and thread placement.

The Jackal compiler is also capable of compiling a Java program for a regular shared memory system, such as an SMP or a Non-Uniform Memory Access (NUMA) system. In this case, the Java program is compiled to machine code without adding object access checks to the binary code. In addition, the DSM runtime is not included in the executable.

### 3.3 Compilation Passes

To support the compilation of OpenMP/Java programs, the JaMP compiler has to extend the existing compiler. Clearly, the OpenMP-enriched Java code has to be mapped to multi-threaded code that is later executed on the target machine. The design of the JaMP compiler and its implementation follow the general architectural principles that are employed to construct OpenMP compilers for other language bindings. With this key requirement defined, it is ensured that the concepts for changing the degree of parallelism in a parallel region of code can be reused in OpenMP compilers for other OpenMP language bindings such as C/C++ and Fortran.

This section first introduces the general architectural features of modern OpenMP compilers (Section 3.3.1). It then describes how the OpenMP/Java
source code is parsed (Section 3.3.2), checked for semantic correctness (Section 3.3.3), and transformed into normal form (Section 3.3.4). The section closes with an overview on the JaMP code generator for OpenMP codes (Section 3.3.5).

### 3.3.1 Architecture of OpenMP Compilers

Figure 3.3 shows the architecture of an OpenMP compiler [127, 197]. Similar to a regular compiler for a sequential programming language, an OpenMP compiler can conceptually be separated into three distinct parts: the front-end, the middle-end, and the back-end. The front-end reads the source code and parses the OpenMP-enriched program. After applying semantic checks, the front-end generates an intermediate representation that reflects the nesting of the OpenMP constructs and directives.
The intermediate representation is passed to the middle-end that transforms it into a normal form. Mapping the OpenMP constructs and directives to normal form saves the back-end from dealing with special cases and simplifies the templates for code generation. The middle-end also might optimize the placement of the OpenMP constructs and directives to make the OpenMP program more efficient. Finally, the back-end generates executable code from the intermediate code representation of the OpenMP program. Before generating the executable code, the back-end splices out the code of the parallel region into a new function that is, later on, executed by the threads when entering the parallel region.

In some compilers, the middle-end is merged with the front-end or the back-end to obtain a simpler architecture. For example, it is possible to create the normal form by applying AST transformation after checking for semantic correctness. It is also feasible to let the back-end perform the transformation to the normal form. We have decided to implement the transformation in the front-end as the AST structure is easier to map to a normal form than the corresponding intermediate code representation.

Following this approach, the Jackal compiler has to be extended at several different places. In our case, the middle-end is merged with the front-end. In the front-end, the parser’s grammar definition has to be augmented with the syntax for OpenMP/Java constructs. In addition, to express parallelization primitives, new node types are incorporated into the existing AST structures. New intermediate code statements are added to high-level LASM that reflect the parallelization structure in the intermediate code. Finally, new back-end compiler passes transform the parallel high-level LASM instructions into the actual parallel code. We now investigate each extension in detail.

### 3.3.2 Parsing

Jackal’s original Java grammar has to be extended to be able to parse OpenMP/Java constructs. We introduce the notion of a `jamp statement` that indicates parallel execution of a Java statement or Java code block that is enclosed in curly braces. Figure 3.4 shows a small excerpt of the OpenMP/Java grammar as we use it for JaMP.

As it can be seen from Figure 3.4, each OpenMP parallel region is considered a regular Java statement; parallel execution is allowed at locations at
3 Implementation of OpenMP/Java in the Jackal Compiler

```java
statement :
   block_statement
   | declaration SEMI
   | expression SEMI
   | "if" LPAREN expr=expression RPAREN statement
     ("else" statement)?
   | forStatement
   | "while" LPAREN expression RPAREN statement
   // other Java statements
   | jump_statement
   ;

jump_statement :
   JAMP_PRAGMA jamp_parallel_construct
   ;

jump_parallel_construct :
   "parallel sections" (jamp_clause)* JAMP_NEWLINE
   LCURLY (jump_statement)* RCURLY
   | "parallel" (jamp_clause)* JAMP_NEWLINE
   | "sections" (jamp_clause)* JAMP_NEWLINE
   LCURLY (jump_statement)* RCURLY
   | "section" (jamp_clause)* JAMP_NEWLINE
   | "num_threads" LPAREN jamp_expr_list RPAREN
   | "shared" LPAREN jamp_ident_list RPAREN
   | "private" LPAREN jamp_ident_list RPAREN
   // ... continue with clauses
   ;

Figure 3.4: Example rules of the OpenMP/Java grammar.
3.3 Compilation Passes

which regular Java statements are permitted. This directly corresponds to the OpenMP specification. Hence, Java’s notion of statements has to be extended with JaMP statements (line 12 in Figure 3.4). As line 16 shows, each OpenMP/Java statement is introduced with a special pragma token (i.e., //#omp, see Section 2.3.2) that is followed by the OpenMP/Java constructs and directive identifiers.

To reduce the grammar’s complexity and to make the resulting parser simpler, constructs and directives are fused into one compound grammar rule. As another simplification, the JaMP’s OpenMP grammar accepts ill-formed nests of OpenMP constructs. For example, from the grammar point of view, it is legal to directly nest a for region in a parallel sections construct; such nestings are disallowed in OpenMP. Such cases are recognized and rejected during semantic analysis of the OpenMP code structure after parsing has finished. This is a well-known technique to keep parsers simple and is applied in almost all existing compilers.

Figure 3.5 shows the AST that is created by the JaMP compiler from the code of the array sum example of Figure 2.2. The lightly shaded nodes of the AST indicate nodes that stem from the regular Java constructs; darkly shaded nodes depict nodes that stem from the OpenMP compilation. As it can be seen, the structure of the OpenMP regions in the program is directly reflected in the AST. The for loop is contained in the parallel for statement and the data-sharing clauses are attached to the root node of the parallel region. The remainder of the AST example then results from the
3 Implementation of OpenMP/Java in the Jackal Compiler

parsing of the for and represents the AST that the existing Jackal compiler would create for the example loop. For simplicity, we omitted the AST of the loop body (indicated by dashed arrows).

3.3.3 Semantic Checks

After parsing, the front-end checks the AST for semantic errors. It is ensured that the OpenMP code is not only syntactically correct but is also semantically correct, that is, the nesting of the constructs and directives corresponds to the OpenMP specification. In addition, expressions that occur in OpenMP clauses attached to constructs or directives are type-checked.

For example, the type-checker only accepts expressions in num_threads that evaluate to an integral value (as prescribed by the OpenMP specification). Similarly, expressions for custom cloners and custom reductions must evaluate to a reference of a type that inherits from the base classes CustomCloner and Reducer, respectively. Finally, data-scoping clauses are inspected to detect inconsistencies such as making a variable both shared and private at the same time.

In this stage, the front-end also verifies that the headers of loops in work-sharing constructs contain valid initialization expressions as well as conforming termination conditions. It also checks for OpenMP-compliant loop-counter update expressions.

3.3.4 AST Transformations

As already mentioned before, an OpenMP compiler transforms the OpenMP constructs and directives to a normal form to simplify code generation for the back-end. Our implementation follows this approach as well and transforms the AST of the program to normal form. The front-end applies AST transformations to the program’s AST and creates a normal form from it.

The JaMP compiler requires each construct and directive to contain a set of clauses that are expected by the JaMP back-end. The front-end always adds num_threads to parallel regions and schedule clauses to work-sharing constructs if not already specified by the programmer. For missing num_threads clauses, the compiler front-end automatically adds a num_threads clause
3.3 Compilation Passes

with an expression to determine the thread count by calling the JaMP run-
time system (i.e., num_threads(JampRuntime.getDefaultTeamSize())). If
a scheduling clause is missing, schedule(static) is automatically added to
a work-sharing construct. Hence, the back-end can rely on always having a
specification for these clauses at the directive level. This alleviates the need
to take care of special cases during generation of the parallel intermediate
code and simplifies the code generation templates.

After the required set of clauses has been added to the AST, the AST is
converted to normal form. For the normal form, compound constructs such as
parallel for or parallel sections (see Figure 3.6(a)) are converted
to a parallel region that directly encloses a for or a sections construct.
A sections construct is then again transformed into a for construct that
contains a switch statement with one case for each section of the sections
construct. A schedule clause with static scheduling with a chunk size of
one ensures that individual section constructs are distributed among the
workers in a round-robin fashion. Figure 3.6(b) shows the resulting AST
as OpenMP/Java source code after the example of Figure 3.6(a) has been
processed by the AST rewriting pass. Again, rewriting sections constructs
to for loops is a well-known technique in OpenMP compilers to reduce the
complexity of the code generators without losing efficiency at runtime.

With the AST transformation applied, the code generation templates that
create the intermediate code out of the OpenMP program only need to sup-
port the parallel construct, the work-sharing constructs for and iterator,
and the barrier and flush directives. All other constructs and directives
have been replaced by their normal forms. Hence, instead of providing eigh-
teen code generation templates for each possible combination of constructs
and directives, the front-end has to support only four templates to compile
every possible OpenMP program to intermediate code.

3.3.5 Code Generation

During Jackal’s regular code generation phase, the OpenMP AST is con-
verted to high-level LASM statements that reflect the structure of the par-
allel code. The intermediate code for the OpenMP example of Figure 2.2
is shown in Figure 3.7 as (unoptimized) high-level LASM code. For sim-
plexity, some instructions such as boundary checks and access checks are
omitted. Each construct is represented as a special intermediate instruction.
(a) OpenMP example with `parallel sections` before AST transformation.

(b) OpenMP example with `parallel sections` after AST transformations applied.

Figure 3.6: Example of the AST transformation for `parallel sections`.
For example, a parallel construct is compiled to a jam-p-parallel-block that contains the compiled code enclosed in the region. Lines 3–6 contain the default num_threads clause that was added during the AST transformation phase. Line 9 represents the work-sharing construct of the for loop. Lines 13–26 contain the body of the for loop. As can it be seen, the loop counter \( i \) (declared in line 10) was privatized as required by the OpenMP specification. Jackal's code generation has replaced the original for loop by a do-while loop guarded by an if statement. This is a well-known transformation in optimizing compilers to save branch instructions and to ease the application of loop-invariant code-motion techniques [7, 178].

Because of the hierarchical structure of the parallel intermediate code, the back-end can analyze and optimize the parallel code in a more sophisticated fashion. If the intermediate code only contained regular intermediate code that uses the Java thread API, the compiler would be limited to standard compiler analysis. Due to the hierarchical intermediate code, the back-end can perform hierarchical Data-Flow Analysis (DFA) [7] and analyze the parallel region together with the surrounding sequential code areas. For example, the JaMP compiler determines the set of variables that are live inside the parallel region to automatically promote variables without scoping clauses to shared variables. Live variable analysis also ensures that a shared variable is only passed to a thread if it is live inside the parallel region, that is, the value of the variable is actually used by the threads. Other optimizations include common sub-expression elimination, bounds check removal, and access check removal.

The JaMP back-end pass eventually converts the high-level parallel LASM code to regular intermediate code that contains explicit calls to the JaMP runtime system that manages the parallel execution at runtime. Depending on the compiler's options, the code generator in the JaMP back-end emits calls to two different kinds of runtime system. The first kind follows the traditional approach of OpenMP compilers that directly map the parallel regions of the OpenMP program to threads (see Section 3.4). The second type of runtime in our system maps the parallel regions of the program to tasks, which are scheduled on a set of system-level threads (see Section 3.5). This approach generalizes the idea of cutting an OpenMP-parallel loop into individual chunks of work (the tasks) to arbitrary OpenMP-parallel codes. We will come back to this type of execution, when reparallelization needs to redistribute work to provide new threads with work (see Chapter 4).
Figure 3.7: High-level LASM representation of the example in Figure 2.2.
The translation of OpenMP’s parallel regions to threads is the most common approach. This approach is followed by most of the other OpenMP compilers and, thus, can be considered standard. Each logical unit of execution is directly mapped to exactly one system unit of execution. Hence, each OpenMP thread is mapped onto one native thread that executes the work of the OpenMP threads. Mapping the parallel regions of an OpenMP program to threads relies on an existing threading library such as POSIX threads [58, 97], the Java thread API [169], or others.

In this section, we provide an in-depth description of how an OpenMP compiler performs this 1:1 mapping of logical UEs to system UEs and, thus, of OpenMP threads to native threads. For the rest of this section, we keep the notion of a thread as a synonym for UE and do not distinguish between LUEs and SUEs, as a 1:1 mapping does not need any distinction. We describe how an OpenMP compiler maps the parallel regions of the to-be-compiled program to a multi-threaded runtime system. In Section 3.5.1 we will show how the compilation for the Java thread API is performed in JaMP.

Most existing threading libraries rely on a single function that is passed to the threads for execution. For example, in the POSIX threading model, a thread is spawned by passing a function pointer to the function that creates the thread. In Java’s thread API, either a class can inherit from the java.lang.Thread class and overwrite the run method, or a class can implement the interface java.lang.Runnable which requires the implementation of a corresponding run method as well [187]. OpenMP compilers traditionally implement this by splicing out the code of a parallel region into a newly created function, a so-called thunk. The thunk accepts a set of arguments in which firstprivate variables and pointers to shared storage locations for shared variables are passed to the thunk.

We now explain how thunks can be created in the JaMP compiler back-end (Section 3.4.1) and how data is passed to the thunk for different kinds of data-sharing clauses (Section 3.4.2). We then provide a description of the transformation templates that are applied to OpenMP loops (Section 3.4.3) and reduction clauses (Section 3.4.4). Section 3.4.5 discusses the implementation of exceptions that arise in parallel regions.
3.4.1 Function Splicing

With *function splicing*, we denote the process of moving a piece of code from one code location to a new function and leaving behind a call at its former location. Function splicing can be thought of as the inverse operation to inlining a function and is also a substantial part of all approaches to procedural abstraction [59]. The general approach to extract arbitrary sequences of code into new functions is described in [118].

Function splicing is needed to create a thunk from the code that is enclosed by an OpenMP parallel region. The OpenMP specification requires parallel regions to adhere to the Single-Entry Single-Exit (SESE) principle [107]. The SESE principle requires that a code fragment is entered and left through only one code path. An example of the SESE principle is a *for* loop without *break* statements. The loop is entered by executing the initialization expression and testing the loop termination condition. If it is true, the loop body is executed and the loop termination condition is evaluated another time. This repeats until the condition becomes false and the loop is exited by branching to the next statement that lexically follows the loop. If, however, the loop contains at least one *break* statement, the loop can be exited on another path by directly jumping out of the loop body with the *break* statement. Hence, the single-exit principle is violated.

Moving the code from a parallel region to a newly created function is accomplished by relocating every single intermediate code instruction to the newly created function. A function follows the SESE property on the low-level of the machine’s execution, as the function can only be entered by executing a call instruction to start the function. After the function has finished, the program is continued right after the call instruction in the caller [16]. A function only violates the SESE principle, when exceptions are thrown during the function’s execution. In this case, the regular compiler’s runtime system has to unwind the call stack to handle the exception [50, 150]. We cover exception-related issues in Section 3.4.5.

3.4.2 Data Scoping

After relocation, the code has to be rewritten to respect the different scopes of variables used inside the parallel region. Whenever a *shared* variable is modified, the new value of the variable has to be written to a shared storage
location that is visible to all threads. If a variable is defined to be `private`, each thread has to receive its own instance of the variable that is only visible to the corresponding thread. For a `firstprivate` or `lastprivate` a private instance of the variable has to be allocated for each thread as well as the value of the variable has to be passed to the thunk or copied out of the thunk, respectively.

OpenMP compilers traditionally solve this problem by adding a set of formal parameters to the thunk. Each parameter serves as a means to transport a certain kind of variable into the thunk or out of the thunk. The OpenMP runtime system loads the various arguments with the current values of the variables that have to be passed to the thunk and executes the thunk. If values have to be passed from the thunk to the sequential code, the runtime system copies the values from the arguments to their corresponding variables in the sequential code.

Figure 3.8(a) and Figure 3.8(b) show an example of how the different data-scoping clauses affect the formal parameters of the thunk. In this figure, we use a C++-like pseudo-code notation to denote pointers as well as Call-by-Reference arguments.

As it can be seen from Figure 3.8(a) and Figure 3.8(b), `shared` variables are assigned to a shared storage location. Each shared variable receives a unique offset into this storage area; the thunk only receives a pointer to the memory area. Finally, all reads and writes to a shared variable are rewritten to reads or writes to the corresponding offset in the shared storage area. On closer inspection, this transformation is not yet complete and correct in all cases. The OpenMP memory model requires a write to the shared storage to be made visible to other threads, when a `flush` is reached. Hence, the compiler has to augment the thunk with memory fences to ensure that writes to the shared storage are made visible to all other threads.

As each thread that is executed in a process receives its own call stack, the most natural solution for `private` variables is to allocate the variable on the call stack. This is done by adding a definition of the variable to the thunk (see Figure 3.8(b)). A `firstprivate` variable is passed as a Call-by-Value argument which transports the initial value of the corresponding variable into the thunk and is automatically allocated a private memory location on the call stack of each thread. Reads and writes have to be redirected to access the newly created formal parameter instead of the local variable that is no longer existent after the code has been moved to the thunk.
Figure 3.8: Pseudo-code example of data-scoping arguments for OpenMP thunks.
3.4 Thread Binding

### 3.4.3 Rewriting of Parallel OpenMP Loops

After the code has been moved to a new function and data accesses have been rewritten to respect the data-scoping clauses, the compiler rewrites any loop that was part of the work-sharing construct. The loop body has to be rewritten such that each thread receives one or more chunks of the loop’s iteration space. The size, the number, and the actual distribution of the individual loop chunks depend on the loop scheduling clause and the type specified therein (see Section 2.3.3).

For a *static* schedule without chunk size, the iteration space is evenly distributed among the workers such that each receives exactly one chunk of the iteration space. If a chunk size is specified, the iteration space is divided into chunks of that size and the chunks are distributed in a round-robin fashion. For a *dynamic* schedule, chunks of equal size are created that are assigned to workers on request, that is, a worker requests a new chunk if it has finished the last one. A *guided* schedule is treated the same way, but the chunk size depends on the number of workers and the number of uncomputed chunks left in the work pile.

Internally, the handling of chunks can be efficiently implemented by means of runtime functions that compute the start and end iteration of a particular chunk. For each scheduling type the runtime system provides a specialized function. Figure 3.9(a) shows an OpenMP-parallel loop with dynamic scheduling and chunk size 5. The code is translated to the thunk that is depicted in Figure 3.9(b).

The structure of the original loop is kept, but the initialization expression and the loop termination condition are changed. The initialization expression no longer starts the loop at the beginning of the loop’s iteration space, but instead starts at \( lb \) as the first iteration of the thread’s current chunk. The same kind of transformation changes the termination condition such that it exits the loop at the end of the chunk.

Depending on the thread ID, the variables \( lb \) and \( ub \) in Figure 3.9(b) are set by the call to `dynamic_chunks` that internally implements the work pile and computes the lower and upper bounds for each requested chunk. The function also returns a boolean value to indicate whether additional chunks are available for the thread. If no chunks are left for the thread to compute, the function returns `false` and the `while` loop terminates.
3 Implementation of OpenMP/Java in the Jackal Compiler

```java
class ChunkDistributionExample {
    void example() {
        int sum = 0;

        // #omp parallel for schedule(dynamic, 5)
        // & reduction(+:sum)
        for (int i = 0; i < 100; i++) {
            sum += compute(i);
        }

        int compute(int iteration) { ... }
    }
}
```

(a) OpenMP loop with dynamic scheduling.

```java
void openmpThunk0(int &global_sum) {
    int lb, ub;
    int sum = global_sum;

    while (dynamic_chunks(thread_id, &lb, &ub, 5)) {
        for (int i = lb; i < ub; i++) {
            compute(i);
        }
        reduce(thread_id, op_plus, global_sum, sum);
    }
}
```

(b) Translation into thunk with calls to runtime system.

Figure 3.9: Pseudo-code example of the translation of OpenMP-parallel loops.
3.4 Thread Binding

For dynamic and guided loops, the work pile only exists on a conceptual level. As the iteration space of the loop is known before the loop is entered, the `dynamic_chunks` function (`guided_chunks` in case of the `guided` clause) can compute the chunk information using counters and arithmetic operations. There is no need to actually store the chunks in a data structure such as a linked list. For static loops (with and without chunk size) the generated thunk only differs in the call to the runtime function `static_chunks`.

3.4.4 Reductions

With reductions, each thread that executes a parallel region can contribute a partial result from its computation to the global result of the computation. Reductions must be performed at the end of parallel regions that have a `reduction` clause associated with them.

An example of a reduction is given in Figure 3.9(a). The initial value of the global `sum` variable is passed to the thunk as a Call-by-Reference parameter. The actual reduction is reflected in the thunk by a call to the runtime system’s `reduce` method. The method expects the current thread’s ID, the type of operation to perform, a reference to the target variable to hold the global result, and the intermediate result of the current thread (see Figure 3.9(b)).

Using such a `reduce` method to perform the reduction step, the runtime system can choose a different reduction algorithm for different numbers of threads. To change the reduction algorithm, the runtime system internally replaces the `reduce` method with another implementation. This can be dynamically performed at runtime by different techniques, such as using `if` statements or even binary code rewriting techniques.

3.4.5 Exception Handling

OpenMP/Java prescribes clean semantics for handling exceptions that arise in parallel regions and that are not handled by exception handlers in the parallel region. Such exceptions are caught by the thread whose execution caused the exception. The thread sets a cancellation flag, registers the exception at the thread team, and terminates if it is not the master thread. The exception handed over to the sequential code is chosen arbitrarily.
Regular Java code already forces a Java compiler to respect exception handling semantics in sequential code. In most Java implementations exceptions are handled by unwinding the stack starting from the method that causes the exception until a corresponding `try-catch` block is reached [50, 150]. Execution continues in the `catch` block of the reached `try-catch`. If an exception caused by a Java thread remains uncaught when the complete call stack was unwound, the Java runtime system supplies a default exception handler that terminates the thread [78].

To comply with the proposed exception semantics of Section 2.3.7, the JaMP compiler can rely on the existing Java exception handling features. Thus, it does not emit special code to parallel regions for handling exceptions. Instead, JaMP implements the exception handling semantics of OpenMP binding for Java in the JaMP runtime system. This keeps both the JaMP compiler simple and the exception handling flexible for future extensions.

Each thread installs an exception handler that catches all exceptions raised in that particular thread (see Figure 3.10). The thunk execution in the thread’s `run` method is wrapped in a `try-catch` handler that matches every possible type of exception (in Java, the common super-type of all exceptions is the `Throwable` class). The handler then executes code that registers the exception, and checks the thread ID to terminate the non-master thread in a controlled fashion.
3.5 M:N Mapping of OpenMP Threads to Native Threads

In its current form, the OpenMP specification does not differentiate between logical UEs and system UEs and it assumes that the OpenMP program is executed on a team of threads (i.e., a team of LUEs). However, it does not specify the layout of the underlying thread execution model. Most implementations use this degree of freedom to map the parallel regions of a program to a native thread library and use a static 1:1 mapping of LUEs to SUEs. This approach was shown in the previous section.

It is, however, possible to separate the notion of an OpenMP thread from the underlying threads that are actually executed on the machine by separating LUEs from SUEs. In this case, \( m \) LUEs that are visible to the programmer (the OpenMP threads) are scheduled to run on \( n \) system UEs. The implementation of Java’s green threads [159] or the multiplexing of threads in single processes [32] follow this approach. Section 3.5.2 explains how multiplexing generally can be implemented.

This novel mapping of OpenMP threads to system-level threads provides another degree of freedom to the OpenMP runtime that the runtime system can exploit to retrofit the OpenMP application onto the computing environment. Since the mapping of LUEs to SUEs is not fixed to a 1:1 mapping, the OpenMP runtime is free to add or remove SUEs as needed while providing the number of LUEs that is requested by the programmer. If, for example, the programmer specifies a high number of OpenMP threads, but the machine only offers a small number of processing elements and a small main memory, running too many SUEs at the same time might exceed the available resources. Moreover, the runtime system can flexibly adapt to changes in the computing environment without the need to change the data and work distribution that is already assigned to the OpenMP threads.

As the performance evaluation in Chapter 7 will show, the Jackal DSM is not able to efficiently handle the more complex data access patterns that originate from the \( m:n \) mapping of OpenMP threads. On a multi-processor shared-memory system, the lower memory access latencies are likely to better support the more complex memory access patterns. In contrast to DSM systems, SMPs provide special machine instructions that can be used to efficiently implement the \( m:n \) mapping. Hence, we can expect a more efficient implementation of the \( m:n \) mapping on SMPs.
We now first describe the key idea of the $m:n$ mapping in Section 3.5.1 before Section 3.5.2 discusses a specialized barrier implementation needed for this mapping.

### 3.5.1 Mapping of OpenMP Threads to Tasks

In the $m:n$ mapping, an OpenMP thread is a task that we regard as a single LUE which has to be scheduled to run on an SUE. Hence, the notion of an OpenMP thread has to be changed to reflect this abstraction. While an OpenMP thread corresponds to a (native) thread in most threaded execution environments, an OpenMP thread matches a task in the $m:n$ mapping. Each LUE of a task is then scheduled to run on one SUE, called the executor thread. A similar approach was investigated by the OMPi project [85] in which several OpenMP threads are run on a (smaller) number of kernel-level threads [192] (the executor threads) to provide a better support for nested parallelism [154].

To maintain correct OpenMP semantics for the $m:n$ mapping, the OpenMP compiler has to create $m$ LUEs for the execution of the parallel region on $n$ SUEs. The exact number of LUEs for the execution of a parallel region depends on the OpenMP thread-number specification (see Section 2.3.5). If more LUEs than SUEs are created, an $m:n$ mapping is automatically performed, that is, $m$ LUEs are automatically mapped to $n$ SUEs and are sequentially executed in them. This blends well with the OpenMP specification, as it does not make any assumptions about the exact interleaving of the execution of individual OpenMP threads. Figure 3.11 shows two example executions of a parallel region (Figure 3.11(a)) with four OpenMP threads each of which is mapped to one LUE. The LUEs are mapped to four SUEs (Figure 3.11(b)) or to two SUEs (Figure 3.11(c)) for execution.

Similar to the OpenMP thread binding, the OpenMP compiler has to transform the code such that it is executed on the tasks created during program execution. To create the tasks, the OpenMP compiler can use the same code transformation templates that have been described in Section 3.4. The compiler splices out the code of the parallel region and moves it to a newly created thunk. During the transformation the templates generate the same code structure that was already depicted in the previous section. The code structure can be kept as it is, since on a conceptual level, each task corresponds to an OpenMP thread that is assigned to exactly one LUE. That is,
3.5 M:N Mapping of OpenMP Threads to Native Threads

```java
class BarrierExample {
    void func() {
        //omp parallel num_threads(4)
        {
            System.out.println("Parallel Region");
        }
    }
}
```

(a) Example of a parallel region executed by four OpenMP threads.

(b) Execution of (a) with four SUEs.

(c) Execution of (a) with two SUEs.

Figure 3.11: Example execution of four OpenMP threads on different numbers of system units of execution.
### 3.5.2 Barrier Implementation in the M:N Model

From a semantics point of view, barriers separate the computation of a parallel region into disjoint phases such that a barrier is only left by a thread if all others have finished computing the last phase as well and have reached the barrier (see Figure 3.12). The $m:n$ mapping requires a different barrier implementation than the implementation in the 1:1 case. Barriers for the 1:1 model block the SUE that enters a barrier until all other SUEs also have reached the barrier. If a barrier in the $m:n$ model did this as well, a deadlock would be caused, as blocked SUEs can no longer execute LUEs that have not been executed so far.

If the parallel region in the example in Figure 3.12 was executed on two SUEs (as shown in Figure 3.11(c)), blocking an SUE due to the barrier in line 6 causes a deadlock, as the OpenMP threads 1 and 2 block the corresponding SUE. The OpenMP threads 3 and 4 will never execute and cause a deadlock at the first barrier. Hence, the SUEs have to be freed as soon as an OpenMP thread is blocked by a barrier. This is necessary, to help avoid deadlocking.
3.5 M:N Mapping of OpenMP Threads to Native Threads

Figure 3.13: Example for LUE splitting of a parallel region with one barrier.

an SUE and to keep the three phases (indicated by print statements) of Figure 3.12 separated.

To free the SUE from the currently executed OpenMP thread and to schedule a different OpenMP thread, the computational state of the current OpenMP thread has to be saved in an efficient way. Two potential solutions to this problem arise: thunk splitting and continuations.

**Thunk splitting**

One way to handle barriers is to split the thunk of a parallel region into several partial thunks. If the parallel region contains \( b \) barriers, the resulting LUEs are split into \( b+1 \) partial thunks, one for each phase (see Figure 3.13(a) and Figure 3.14(a)). Reaching a barrier, one partial thunk must be replaced by another of the same computational phase. While this approach maintains correct barrier semantics, it cannot be actually implemented, as data dependencies between partial thunks of the same LUE cannot be correctly reflected by this naive scheduling approach.

However, we can extend the naive approach to correctly reflect data dependencies. All values of all live variables need to be transported between the executions of the partial thunks that belong to the same LUE. Passing the values of live variables as an argument to the thunk, thunk invocations can implicitly maintain data dependencies by placing stack frames on the call stack of the executor thread.
Figure 3.13(a) shows an example of three LUEs that are split into two thunks “T_n-A” and “T_n-B” because of the barrier. To free the executing SUE from the current thunk, the currently executed thunk invokes a thunk of another LUE (solid arrows represent function calls; dashed arrows indicate returns from function calls). For example, “T_2-A” invokes “T_3-A”. After the last partial thunk “T_3-A” reaches the barrier, “T_3-A” calls “T_3-B”, passing along the values of live variables at the barrier. Eventually, this leads to the call graph of Figure 3.13(b). Barrier semantics are preserved, as partial thunks “T_n-A” of phase 1 execute before the thunks of phase 2 are invoked.

While data dependencies are maintained, this approach is only applicable for parallel regions with only a single barrier. Figure 3.14(b) shows the recursive calls that originate from the region with two barriers (in Figure 3.14(a)). With the recursive calling scheme presented above, it is impossible to execute the partial thunk “T_2-C” at the right time. As “T_2-B” has preserved the values of live variables for “T_2-C”, “T_2-C” has to be executed as a callee of thunk “T_2-B”. However, correct execution semantics prescribe that “T_2-C” is scheduled for execution after all thunks “T_n-B” have been finished. As a stack frame of a function is destroyed after the function has terminated, live variables needed for T_2-C cannot be preserved (e.g. Icon [79] provides stack-preserving functions at the expense of performance). Evaluating all possible execution schemes for the partial thunks, it becomes clear that no recursive execution scheme exists that can correctly preserve barrier semantics. Hence, the thunk splitting approach is rendered unusable for a barrier implementation supporting any number of barriers in the \( m:n \) mapping.

Figure 3.14: Counter example for LUE splitting of a parallel region with two barriers.
Continuations

Continuations that free an SUE from the current LUE solve the scheduling problem in a natural way. A continuation represents the control state of a computation at a given point of the computation (i.e., program counter, live variables, call stack, etc.) [207]. The continuation implicitly also records the intra-LUE data dependencies. In system programming, the continuation is called Process Control Block (PCB) and postponing an SUE (i.e., a kernel-level thread) on the processing element is called a context switch between kernel-level threads [173, 192]. The PCB, however, not only conserves the contents of registers, but also stores additional information needed by the operating system such as memory limits, list of open files, virtual memory tables, etc. In our case, the computational state is described by less data and thus the continuation can be kept smaller than a PCB.

If an LUE reaches a barrier, a continuation of the LUE is created and stored in a list. As the LUE’s further execution is postponed, the SUE is now free to process the next LUE of the same computational phase. Once all LUEs have finished computing the current phase, each SUE resumes one LUE and continues computation; the rest of the LUEs wait until either the next user-defined barrier or the implicit barrier at the end of the parallel region is reached. Although similar approaches multiplex user-level threads on a set of kernel-level threads [52, 75], none supports malleability by adding or removing kernel-level threads as needed.

Postponed continuations are stored in a linked list that is allocated for each active parallel region and that is organized as a First-In First-Out (FIFO) queue. Hence, LUEs that are stored in the list are retrieved in the same order as they were inserted. In addition, if the list is empty, the SUE blocks until an element is appended to the list. The list may become empty if the number of LUEs is smaller than the number of SUEs. To ensure barrier semantics, for each continuation a counter is maintained that identifies the current execution phase. Due to the FIFO property and the use of the phase counter, only continuations whose counters match the ID of the currently active barrier are permitted to be retrieved from the linked list of continuations. If all LUEs have reached the barrier (and thus have the same counter value), the counter is incremented and LUEs are fetched from the list and assigned to a free SUE. This continues until the implicit barrier of the parallel region is reached and the parallel region exits.
A closer look to the state changes that occur in the implementation of barriers for LUEs and its associated linked list reveals that not all LUEs have to be postponed by creating and storing continuations to correctly implement the semantics of a barrier. If for each SUE at most one LUE is left in the queue of postponed LUEs for execution, the SUEs can use a regular counting barrier to block and wait for the other SUEs to reach the barrier as well. Switching to a regular barrier implementation lowers the runtime overhead of entering and leaving the barrier, as counting barriers or more optimized versions (such as static $f$-way tournament barriers [83] or dissemination barriers [89]) are much cheaper to implement than the multiplexing barrier based on continuations.

The example of Figure 3.15 illustrates this optimization. Suppose four logical UEs with a single barrier have to be executed two system units of execution. For the first two logical units of execution, the runtime system creates a continuation for each LUE that reaches the barrier and appends it to the list of postponed continuations. Obviously, each SUE has to process at most
3.6 OpenMP/Java Runtime Implementation in Java

The previous sections have shown the code templates for two different kinds of execution models. In Section 3.4, a 1:1 mapping of OpenMP threads to executor threads has been described. We then generalized this mapping to an \( m:n \) mapping in which several OpenMP threads can be considered as LUEs that are multiplexed onto SUEs for execution (see Section 3.5).

The compiler transforms the code of a parallel region into a thunk containing the original code and augments the thunk with runtime system calls that control the parallel execution of the thunk. For the 1:1 mapping, the runtime system creates the thread team, passes the thunk reference, performs reductions, and waits for the completion of the execution. For the \( m:n \) mapping, the runtime is also schedules the LUEs on the SUEs during execution. For both bindings, the JaMP runtime has to provide a full runtime system. Features of the runtime system include thread and task handling, a barrier implementation for threads and tasks, mechanisms to handle exceptions thrown in parallel regions, etc. We will now describe the runtime system that is used by our compiler to implement JaMP. Traditionally, three solutions arise for an implementation of the runtime system.

First, the compiler can generate the runtime system by emitting intermediate code for it. However, this implies that the functionality of the runtime system has to be implemented as templates in the code generator of the compiler. This makes the code generation phase unmanageably complex.

Second, the runtime system can be written in another language such as C/C++. In our case, C is the only option, as parts of the Jackal runtime system already are coded in C. Interaction with the existing Java API becomes a problem, since each call to Java code has to be performed through the Java Native Interface (JNI) [125]. This would decrease performance and would add unnecessary complexity for handling the JNI conversion to the runtime system code.
Third, the OpenMP/Java runtime system can fully be implemented in Java. This enables a smooth integration into the Java API and benefits from Java’s high-level programming model. Hence, for obvious reasons, we entirely implement JaMP’s runtime system in Java. While its implementation mainly focuses on clarity and maintainability, care has been taken to provide an efficient implementation.

Section 3.6.1 describes the implementation of the JaMP runtime for the thread binding. Section 3.6.2 shows how OpenMP-parallel loops have to be rewritten for execution by the JaMP runtime. In Section 3.6.3 the JaMP runtime for the \( m:n \) mapping with Java tasks is presented.

### 3.6.1 The OpenMP/Java Runtime for Java Threads

As already mentioned above, the JaMP runtime for the 1:1 model executes a parallel region with a team of Java threads that constitute the executor threads. Each SUE executes exactly one LUE, that is, the OpenMP thread that executes the thunk. As Java neither provides function pointers (as needed to pass a reference to a thunk) nor defines any means to pass Call-by-Reference parameters or pointers to shared storage, thunks cannot be passed directly to the threads for execution. Thunks and data passing features have to be emulated with pure-Java constructs.

In the following, we will first investigate how thunks of parallel regions can be reflected by using Java’s object-oriented features. We then focus on the implementation of data-scoping clauses and describe how thunks can be executed by Java threads.

**Implementing Thunks in Java**

To simulate the necessary features, we employ the concept of functors [47] and closures [1]. A *functor* is a special kind of class whose instances behave like functions. Functors are used in object-oriented languages to provide a type-safe equivalent of function pointers that exist in other languages (e.g. C). In C++, functors are implemented as classes that override `operator()` [109]. In Java, functors typically contain a single `execute` or `run`
method that describes the computation of the function to perform. A common example of functors in Java are objects that implement the Runnable or the Callable interfaces.

As a closure we denote an expression or function that encapsulates (parts of) the execution environment in which the expression or the function has to be evaluated. The concept of a closure is well-known in functional programming and is used to conserve the values of variables in deferred function evaluations. In Java programs, closures are used to augment functor objects with an internal state that is used during the call to the functor.

In the JaMP runtime, thunks are implemented as a special thunk class that inherits from the JampRunnable base class (see Figure 3.16(a)). This class plays the role of both a functor and a closure object, providing an abstract method execute. For the sake of compatibility, JampRunnable also implements the Runnable interface to make the functor of an OpenMP/Java thunk compatible with Java’s notion of functors and the interfaces of the Java threading API (see below). Inheriting from JampRunnable, the compiler-created thunk class overrides the execute method with the thunk code generated at compile time (see Figure 3.16(b)).

Data-scoping

The JampRunnable class also plays the role of a closure to transport values of variables into the thunk and from the thunk back to the sequential code. As closure state, it uses two instances of a special container class (JampParam) that can store an arbitrary number of values. One instance of this class (shared, Figure 3.16(a)) is used to provide the storage area for shared variables. Another instance (.private) stores any kind of privatized variables, that is, variables that are marked as firstprivate, lastprivate, and reduction. However, reduction variables are treated in a special way. While each thread owns a private incarnation of the variable, a shared incarnation is also created by the compiler. This shared incarnation is used to hold the final result of the reduction and to transport this result back to the sequential code.

A private variable is not stored in the .private instance, as these variables can be allocated on the call stack of a thread which makes them automatically private to this thread. As the initial value of a private variable
3 Implementation of OpenMP/Java in the Jackal Compiler

```java
abstract class JampRunnable implements Runnable {
    private Object _this;
    private JampParam paramShared;
    private JampParam paramPrivate;
    private JampThreadTeam threadTeam;
    private int startLabel;
    private int lateEntry;

    public void run() {
        JampThread thread =
            (JampThread) Thread.currentThread();
        execute(_this, paramShared, paramPrivate,
            thread, threadTeam, thread.getId(),
            startLabel, lateEntry);
    }

    abstract void execute(Object thisObject,
        JampParam shared,
        JampParam _private,
        JampThread threadObject,
        JampThreadTeam threadTeam,
        int threadId, int startLbl,
        int lateEntry);
}
```

(a) Wrapper object for executing parallel regions.

```java
class Thunk_0 extends JampRunnable {
    void execute(Object thisObject,
        JampParam shared, JampParam _private,
        JampThread threadObject,
        JampThreadTeam threadTeam,
        int threadId,
        int startLabel, int lateEntry) {
        int pr = 0; int lp = 0;
        int fp = _private.getInt(0);
        System.out.println(shared.getInt(0) //sh1
            + shared.getInt(1)//sh2
            + pr + fp + lp);
        _private.setInt(1) = 1; //lp
    }
}
```

(b) Thunk class generated for the thunk in the example of Figure 3.8.

Figure 3.16: Functor and closure to implement OpenMP/Java thunks.
corresponds to Java’s default value for the particular type of the variable, they can be initialized with their respective default values upon declaration. Hence, passing a value in the _private container is not needed.

As it can be seen from Figure 3.16(b), reads and writes to variables are redirected by the compiler to their respective storage locations. The compiler statically assigns an offset for each variable that is stored in an instance of JampParam. In the example of Figure 3.16(b), the original variable sh1 (see Figure 3.8) is assigned offset 0 in shared, while sh2 is assigned an offset of 1. While the assigned offsets are unique, the same offsets are allowed to occur for different primitive types. Privatized variables are treated the same way; each of them receives a unique offset in the _private container to retrieve or store the value of the original variable.

**Thunk Execution with Threads**

As Java already contains language constructs for parallel programming, it seems natural to directly use the Java threading API [147] to provide for the parallel execution environment. However, the very limited basic features of Java threads are not enough for the JaMP implementation, as there is no support for thread pools and the exception handling in threads does not fit the requirements of OpenMP/Java. While being grouped into so-called ThreadGroups, a regular Java thread can hardly be assigned to a team of threads and controlled by collective operations. In addition, Java threads do not offer the concept of thread IDs and distinguishing a thread from another thread requires an expensive call to Thread.currentThread(). Hence, the JaMP runtime has to extend the notion of a thread (and of thread groups) and add the desired features.

The foundation for a JaMP thread is the java.lang.Thread class that can be used in two complementary ways. First, a class that inherits from Thread can be created. The class provides the code to be executed by overwriting Thread’s run method that is invoked after the thread is started by a call to its start method. Second, the constructors of Thread accept an object reference to an object whose class implements the Runnable interface. The interface requires to implement a run method that contains the code that should be executed by the thread. The Thread object saves the reference passed at construction time and calls the run method upon thread start.
The JaMP implementation intermixes both forms of thread construction to internally provide a high-level threading facility for OpenMP/Java. It first creates a new kind of thread (the JampThread) by inheriting from Thread and overwriting the pre-defined run method such that it executes objects of classes that inherit from the JampRunnable class and contain the code of a thunk and its execution environment. Additionally, the run method also implements OpenMP/Java’s exception handling semantics by setting the cancellation flag and registering the exception at the thread team (see Section 3.4.5).

At the original location of the OpenMP/Java construct, the compiler inserts calls to create the thread team, to start parallel execution, and to wait for the completion of the parallel execution. Figure 3.17 shows the Java code that is inserted as intermediate code during compilation.

Parallel execution of the example code of Figure 3.17 is prepared by determining the number of threads for the execution (line 4). As the parallel clause does not specify num_threads, the JaMP compiler inserts a call to a default method that determines the thread count (see Section 2.3.5). This value is later passed to the thread team.

The instances of JampParam objects for private and shared variables are created in lines 5–8. The first instance (tmpl) is used as a template that is later used by the thread team to construct the private JampParam objects for the individual threads. The second instance (vars) transports the values of the shared variables array and sum into the parallel region and back. Although sum is a reduction variable and thus has to be privatized, a shared incarnation exists to transport the final result of the reduction after the parallel region has ended (see Section 3.6.1). In lines 9–10, the offsets assigned to the variable (argument one of the setInt method) are used to store the value of the variable in the JampParam instance. The compiler assigns two times the offset 0, as the variables sum and array are of different types.

After the JampParam objects have been created and the value of variables have been stored, the thread team is constructed and parallel execution starts (line 11–14). Upon creation, the tread team receives the thread count, the references to the JampParam objects, the this reference of the surrounding method (null in case of a static method), the number of barriers in the parallel region, and a string that contains the class name of the generated thunk. After the startAndJoin method returns, lines 15–16 update the shared local variables; sum then contains the result of the reduction.
3.6 OpenMP/Java Runtime Implementation in Java

```java
class SumArrayExample {
    int sumArray(int[] array) {
        int sum = 0;
        int nthr = JampRuntime.getDefaultTeamSize();
        JampParam tmpl = JampParam.createJampParam(...);
        JampParam vars = JampParam.createJampParam(...);
        vars.loadInt(0, sum, JampParam.SHARED);
        vars.loadInt(0, array, JampParam.SHARED);
        JampThreadTeam team = JampThreadTeam.createTeam(nthr, tmpl, vars, this, 1, "Thunk_0");
        team.startAndJoin();
        sum = vars.getInt(0);
        array = vars.getInt(0);
        return sum;
    }
}

class Thunk_0 extends JampRunnable {
    void execute(...) {
        int sum = 0;
        int[] array = (int[]) shared.getObject(0);
        LoopInformation loop = team.getLoopInformation();
        process_next_chunk:
            int chunkId = loop.getStaticChunkId(...);
            if (chunkId == -1) goto finish;
            int i = loop.getFirstIteration(chunkId);
            int end = loop.getLastIteration(chunkId);
            if (i<end) {
                do {
                    sum += array[i];
                    i++;
                } while (i<end);
            } goto process_next_chunk;
        finish:
            team.addToReductionPlus(threadId, 0, sum);
        }
}
```

Figure 3.17: Pseudo-code generated for `sumArray` in Figure 2.2.
After creating the thread team, each thread of the thread team calls the `execute` method (line 22) to execute its share of the parallel region. The method receives a reference to the executing thread to be able to easily determine the executing thread and to perform actions on it (such as notifying the thread about an exceptional event). As an optimization, the thread ID is also passed to the method to save an expensive call to the thread object in case the ID is needed (e.g., to determine if the current thread is the master thread). Finally, the `threadTeam` parameter contains a reference to the thread team. A thread needs to know about the team it belongs to so that it can query the runtime system for custom cloners, user-defined reduction operators, setting the cancellation flag, etc.

### 3.6.2 Rewriting of Parallel OpenMP Loops

Section 3.4.3 has shown how an OpenMP compiler translates OpenMP loops such that they obey the scheduling requirements that were specified by the programmer. Conceptually, the loop is encompassed by a second loop that queries the runtime system for the boundaries of the next chunk that is to be processed by the translated loop. If no chunks are available, the outer-most loop terminates.

In the JaMP runtime, this call is reflected by a `LoopInformation` object that contains information about the iteration space of the to-be-parallelized loop. For each kind of scheduling, the `LoopInformation` object offers calls to methods that internally compute the boundaries of the next chunk that is assigned to the currently calling thread. The `LoopInformation` object is created only once and is shared among all workers of the current team. Synchronization on the calls to the methods of the `LoopInformation` object is only needed in case of dynamic or guided scheduling, as these need concurrent updates to the state of the conceptual work pile.

Figure 3.17 shows pseudo-code for the code that our JaMP compiler generates. First, the current chunk ID is determined (line 28). If the ID equals -1 all chunks have been processed and processing of the work-sharing construct stops (line 29). Depending on the scheduling type, the compiler inserts calls to different methods of the `LoopInformation` object. In case of static scheduling, `getStaticChunkId` is used. The method accepts as arguments the number of to-be-processed chunks, the thread count, and the thread ID of the executing thread. Using these values, the method computes which
chunk has to be computed next. In case of a dynamic scheduling strategy, the compiler inserts a call to `getDynamicNextBlockId` without arguments. Conceptually, all uncomputed chunks are stored in the work pile. Each call to `getDynamicNextBlockId` removes one chunk from the pile and returns its ID to the calling thread.

Second, lines 30–31 determine the partial iteration space of the current chunk by querying the loop-information object. Depending on the scheduling type, the methods `getFirstIteration` and `getLastIteration` compute the lower and upper bound of the loop counter for the current chunk. Except for guided schedules, these methods directly compute the boundaries by multiplying the chunk ID with the chunk size. For a guided schedule, the methods also consider the uncomputed chunks, as the chunk size depends on the amount of unprocessed work [154].

Finally, the original loop is executed. As the loop counter is set to the lower bound of the chunk and a temporary variable contains the upper bound, the loop only processes the current chunk and exits. Line 38 jumps to the outermost loop at which the next chunk is requested for computation. Separating the loops keeps the work-sharing loop simple enough to trigger Jackal’s loop optimizations, whereas fusing both loops would create a harder-to-optimize version of the code.

Decomposing the single call of the method that computes the chunk into several small function calls does not negatively impact performance. First, the overall computation of the single functions is the same as in the single-function case. Second, with inlining enabled, the Jackal compiler back-end directly inserts the body of the smaller methods into the outer loop and therefore avoids the overhead of calling these functions.

**Reductions**

The values of reduction variables are transported the same way as variables marked as `firstprivate` and `lastprivate` are passed into and out of a parallel region. To transport the intermediate result, the compiler creates code that stores the result into the `.private` field of `JampRunnable`. This transformation is performed, when the compiler rewrites variables to respect the data-scoping clauses of the translated parallel region.
To perform the reduction, OpenMP compilers traditionally add a function to the end of the parallel region. In the JaMP runtime, it is the thread team that provides the functions for all kinds of reductions as instance methods. It also provides a generic reduction method that expects an object that implements the `Reducer` interface (see Figure 2.10). See line 40 in Figure 3.17 for a pseudo-code example of how the reduction of Figure 2.2 is translated. The reduction variable was assigned ID 0 in this case and the partial result is passed by the local variable `sum`.

In the current version, only a sequential reduction algorithm is implemented for simplicity. However, a tree-based reduction algorithm can be added by changing the internal implementation of the reduction methods of the thread team. As each thread already calls the method from within the parallel region, the method could internally provide the logic to reduce the results in parallel as well. However, this is left open for future implementations of OpenMP/Java in Jackal.

### 3.6.3 The OpenMP/Java Runtime for Java Tasks

In this section, we describe our implementation of an \( m:n \) mapping of logical OpenMP/Java threads to native threads. In this mapping, a set of logical OpenMP threads is scheduled to run on a (smaller) number of SUEs, the executors. The generic implementation strategy has already been shown in Section 3.5. Let us now focus on the Java implementation of this mapping.

Since version 1.5, Java contains the `java.util.concurrent` API for high-level parallel programming. The Java thread API relies on threads to express parallelism, which makes parallel programming cumbersome and error-prone. With the new API, it becomes possible to easily express asynchronously executed code by means of `task` objects. As tasks are automatically mapped to a set of threads for execution by the Java runtime system, the programmer is freed of directly implementing the parallel code in user-defined threads. By submitting tasks to a so-called `executor service`, the execution strategy can be adopted to the executing environment by replacing the executor service only. For example, if only a sequential execution is desired, a multi-threaded executor service can be (automatically) replaced by a sequential executor service without affecting task creation and submission in the rest of the program. Hence, developing a program for different kinds of platforms is made easier as the degree of parallelism can
be easily adapted by changing the executor service while keeping the rest of the program unchanged. In addition, the API provides new high-level synchronization concepts such as ReadWrite locks, an efficient implementation of atomic data types (e.g. AtomicInteger), and data structures that are optimized for efficient concurrent use (e.g. PriorityBlockingQueue).

The rest of this section first explains the application of Java tasks of the java.util.concurrent package by means of a short example. It then describes how the OpenMP/Java thunks can be mapped to JaMP tasks for execution. Finally, the JaMP executor service that actually executes the JaMP tasks is presented.

**Asynchronous Computations with Tasks**

Asynchronous execution of Java code is accomplished by creating an instance of a FutureTask object that requires an object of type Callable at construction time. Similar to Runnable, Callable plays the role of a functor interface with a prescribed method call that takes no arguments. Contrary to Runnable’s run method, call may return an object reference as its result. Adhering to the concept of functors, asynchronously executed code is moved to a user-defined class that implements Callable and provides an implementation of the call method. If values have to be passed to the call method, the class has to declare instance variables that are used to store the to-be-passed values. The code is then passed to a so-called executor, that wraps the code in a future object (FutureTask) and schedules the created future object for execution on a thread of the executor. To query the result of the asynchronous computation, FutureTask provides a get method that waits until computation has finished.

As an illustrative example, Figure 3.18 shows a task-based version of our running example from Figure 2.2. While the original example computes the sum of the array only, we have extended the code of Figure 3.18 to concurrently compute the product of the array while the array is processed for calculation of the sum. The code for the array sum is moved to a class that implements Callable (lines 6–17). As required, the call method is implemented in lines 11–16 to compute the sum of the array elements into a local variable, to wrap it in an Integer object, and to return the result. To access the array elements, the AsyncSum class needs to store a reference
3 Implementation of OpenMP/Java in the Jackal Compiler

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.FutureTask;

class AsyncSum implements Callable<Integer> {
    private int[] array;
    AsyncSum(int[] array) {
        this.array = array;
    }
    public Integer call() throws Exception {
        int sum = 0;
        for (int v : array)
            sum += v;
        return new Integer(sum);
    }
}

class ArraySum {
    public static void main(String[] args)
        throws Exception {
        int[] array = new int[10];
        for (int i = 0; i < array.length; i++)
            array[i] = i + 1;
        AsyncSum as = new AsyncSum(array);
        ExecutorService es = Executors.newFixedThreadPool(4);
        FutureTask<Integer> ft = es.submit(as);
        int product = 1;
        for (int i = 0; i < array.length; i++)
            product *= array[i];
        int sum = ft.get();
        System.out.println("sum " + sum +
            " , product "+ product);
        es.shutdownNow();
    }
}
```

Figure 3.18: Example of an asynchronous computation of an array sum.
to the array. Hence, containers for asynchronous code are a form of user-implemented closures, as they not only contain the code to execute, but also transport the environment in which the code has to be evaluated.

The main method in Figure 3.18 first creates the input array (line 22) and initializes the elements with a growing sequence of integer numbers. Lines 26–29 create an instance of the AsyncSum closure class and wrap it in an instance of FutureTask that is then passed to the executor service in line 29. For simplicity, the example’s executor service uses a fixed-size pool of four threads. For more complex scenarios in real world applications, Java’s java.util.concurrent API also offers growing and shrinking thread pools with time-outs, etc. After execution of line 29, an asynchronous computation runs in one of the executor threads in the background and the main thread concurrently continues with the calculation of the product of the array elements (lines 31–33). Eventually, in line 35, the result of the asynchronous task is needed and the main method waits until the background computation has finished and the result is returned.

Implementing OpenMP Threads with Java Tasks

Instances of JampRunnable are used as functors that are evaluated in a closure to implement thunks for the Java thread implementation. As the JampRunnable interface is conceptually already very close to the interface of the Callable interface, the m:n mapping can exploit the executor service API as a provider for SUEs. We can use Java tasks as LUE implementations and wrap JampRunnable objects in Callable objects to satisfy the requirements of the Java task API. Thus, most parts of the runtime system can be kept without changes to the API and, most importantly, without changes to the code generator (except using a different barrier implementation). Hence, everything that was described in Section 3.6.1 is re-used in the compiler back-end to compile the OpenMP-parallel code fragments.

The overhead introduced by re-using the existing compiler back-end and the internal APIs of the runtime systems is minimal. The only overhead incurred is the creation of one additional object (the task) that contains a reference to the JampRunnable instance and an additional method call, as the task only redirects the control flow to the wrapped object. However, we accept this minor performance penalty, as this approach keeps the structure of the runtime system clean and easy to maintain.
### OpenMP/Java Executor Service

The tasks that are created for the execution of the LUEs of a parallel region have to be handed over to an executor service that actually maps the tasks to the executor threads for computation. At first glance, it seems possible to rely on the Java’s default executor services. However, these are only suited for tasks that are executed in one go, that is, tasks that are scheduled for execution, compute until finished, and return a result. Postponing tasks is not supported by the regular Java API. Since the LUE tasks contain barrier code that potentially postpones tasks as a continuation, the JaMP execution model does not fit the execution model of the `java.util.concurrent` API.

Because of this, the JaMP executor service is based on Java’s implementation of the `AbstractExecutorService` in the `java.util.concurrent` package and extends the default functionality. Where necessary, it extends the abstract executor with the functionality needed to execute and postpone JaMP tasks (see Section 3.5.2). The JaMP executor service could fully replace the Java executor services and is not limited to executing OpenMP/Java tasks, as it also satisfies the requirements to execute regular Java tasks.

Our executor service supports continuations to free an executor thread from an OpenMP/Java task and resume it later on. The executor therefore contains a completion queue, in which the tasks are stored when a barrier is encountered (see Section 3.5.2). The executor relies on the cooperation of the task to create the continuation: The task saves its live variables into a `JampParam` object, calls its `setLiveVars` method to store its live variables in the continuation, and then returns passing along a flag that is set if the task is finished or unset if it is not finished. The executor evaluates the flag and inserts the task into the completion queue with an incremented execution phase if necessary, that is, if the task has not finished computation. Execution automatically ends when the tasks have reached the end of the parallel region and the task objects return from their `call` method.

#### 3.7 Summary

This chapter has shown how JaMP implements the OpenMP/Java binding in the Jackal DSM system. To provide a solid base for the reparallelization
features that are described in the next chapter, we apply generic design principles for our implementation of OpenMP for Java.

At first, we have investigated traditional approaches to design and implement OpenMP compilers. Starting with syntactic and semantic analysis, we illustrated all phases of an OpenMP compiler. AST transformations convert OpenMP code to normal form to keep the code generation templates small. Reparallelization benefits from normal form, as the number of constructs that have to be supported is restricted to parallel regions and work-sharing constructs only. We will revisit this topic in the next chapter.

We then have presented the implementation of a traditional thread binding for the execution of OpenMP programs. In this binding, the compiler creates so-called thunks that contain a region’s code for parallel execution. The code generation templates move the code from its original location to the created thunk and rewrite the moved code such that it respects the data-sharing clauses specified at the directive level. A work-sharing construct is rewritten such that each thunk executed in parallel only processes a partition of the formerly sequential loop.

A contribution of this chapter is the \( m:n \) task-based mapping of OpenMP threads to SUEs, the executors. With this mapping, the executor service of the runtime system decouples the execution of LUEs on SUEs. As LUEs and SUEs are decoupled, the runtime gains another degree of freedom by adjusting the number of SUEs depending on the resources available in the computing environment. As we will see in the following chapter, the multiplexing approach uses this degree of freedom to implement malleable applications.

JaMP yet only supports OpenMP/Java as described in Chapter 2. The OpenMP 3.0 task construct can easily be added to the described execution models. In the threaded model, tasks are stored in a queue and the executors withdraw uncomputed tasks from the queue. In the \( m:n \) model, a task queue already exists to hold continuations and, thus, can be re-used. Other OpenMP 3.0 features can also be easily added to JaMP.

Finally, this chapter has provided a discussion of the OpenMP/Java runtime system. For maintainability, the runtime is entirely implemented in Java. It is designed in an object-oriented fashion and allows for a maximum of flexibility by providing interfaces to alter aspects of the implementation without recompilation. However, care has been taken to find a tradeoff between flexibility and efficiency.
Applications that can automatically alter their degree of parallelism during execution are called malleable (see Section 1.2). If changes to the computing environment occur, that is, processing elements are added to or removed from it, the application is notified about the change. It then adds or removes UEs as needed by the new number of processing elements. It depends on the implementation of malleability whether the number of logical units of execution or the number of system units of executions is altered.

This chapter focuses on reparallelization as one solution to make an application malleable. In the last chapter, an OpenMP implementation was shown that followed the general design and implementation techniques that is used for OpenMP compilers and runtime systems. Hence, our reparallelization approach is applicable to all OpenMP implementations. We use our OpenMP implementation as a basis and augment it with reparallelization features to alter the number of LUEs that execute parallel regions.

Our reparallelization approach for OpenMP is based on the following requirements. First and most importantly, a reparallelization approach has to cover all OpenMP constructs. Because of the OpenMP compiler’s normal form for OpenMP constructs (see Section 3.3.4), reparallelization features need to be implemented for work-sharing constructs and parallel regions only. The second requirement is efficiency. Adding reparallelization features to the OpenMP application must not cause a severe performance drop for application execution. Algorithms and data structures have to be chosen carefully such that they do not introduce high overheads during normal execution of the application. Third, the approach has to be non-invasive, that is, the programmer should not need to change the program code to enable
the reparallelization features for the application. Reparallelization should be transparently enabled by the compiler instead. Finally, if there are limitations to the programming model, these limitations should be kept as small as possible.

This section is organized as follows. Section 4.1 provides an overview of the state of the art in making programs malleable. In Section 4.2, we highlight the potential approaches and discuss advantages and drawbacks. Section 4.3 discusses a compiler’s ability to deduce the work and data partitioning of a parallel application. Section 4.4 and Section 4.5 show how the various OpenMP constructs can be reparallelized. Limitations of the reparallelization approach covered are discussed in Section 4.6. Finally, Section 4.7 presents the compiler templates that are needed to compile reparallelizable OpenMP applications.

4.1 State of the Art

We are not aware of any approach for reparallelization of OpenMP applications, but projects targeting malleability of (OpenMP) applications exist. Before investigating the state of the art in malleable OpenMP applications, this section investigates malleability in two other wide-spread programming models, that is, multi-threading APIs and MPI.

Table 4.1 lists the programming models and related projects that are discussed in this section. For these parallel programming models, the table applies the classification scheme of Figure 1.4 and categorizes the programming models accordingly. The tag “✓” indicates a full match in the corresponding category. The “✗” symbol denotes that the programming model does not match the category. If a “●” is depicted, the programming model does not fulfill all of the above requirements, but can be considered for the category, as the programming model partially complies with the requirements of the respective category.

Table 4.1 lists the programming models and related projects that are discussed in this section. For these parallel programming models, the table applies the classification scheme of Figure 1.4 and categorizes the programming models accordingly. The tag “✓” indicates a full match in the corresponding category. The “✗” symbol denotes that the programming model does not match the category. If a “●” is depicted, the programming model does not fulfill all of the above requirements, but can be considered for the category, as the programming model partially complies with the requirements of the respective category.

All applications written in one of the mentioned programming models can be considered rigid, because the programmer can always restrict the application’s execution to a determined degree of parallelism and refuse execution if not enough PEs can be provided by the execution environment. Moreover, all projects support evolving applications.
Table 4.1: Classification of malleability in different programming models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Rigid</th>
<th>Evolving</th>
<th>Moldable</th>
<th>Malleable</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX threads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Java threads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Java concurrent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>MPI</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MPI-2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>PVM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>GrADS</td>
<td>✓</td>
<td>●</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>[133]</td>
<td>✓</td>
<td>●</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>[69]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMPI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cactus</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>DGET</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>OpenMP</td>
<td>✓</td>
<td>●</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>[191]</td>
<td>✓</td>
<td>●</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>[131]</td>
<td>✓</td>
<td>●</td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>JaMP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Legend:
✓ full match
✗ no match
● partial match
4.1 State of the Art

The low-level threading APIs (e.g. POSIX [58, 97] or Java threads [147]) do not provide features to automatically change the number of threads while the program is running. However, as thread creation and termination is under control of the programmer, it is possible to implement evolving applications by using the corresponding API functions that add or remove threads. The programmer is responsible for detecting changes in the computing environment and for creating or terminating threads as needed. A compiler cannot analyze the threaded code to adapt the thread count automatically (see Section 4.3). JaMP’s malleability feature provides an automatic way and automatically adapts the degree of parallelism for OpenMP applications.

Java’s java.util.concurrent package [187] offers executor services to the programmer (see Section 3.6.3). With the ThreadPoolExecutor executor service, threads that execute the submitted tasks may be created and terminated on demand. The number of threads depends on the caching policy and is not determined by examining the resources available in the computing environment. Future versions of Java might incorporate policies that respect changes in the environment. JaMP automatically handles the assignment of OpenMP threads to executors and reacts on any change in the computing environment.

Most MPI [142] applications are designed to be moldable, that is, the number of PEs is determined at start-up and remains fixed during runtime. With MPI-2 [143] and PVM [190], it is possible to implement evolving applications that alter their degree of parallelism during execution. MPI-2 defines the MPI_Comm_spawn primitive that creates a new MPI process and adds it to the application. In [123], a hybrid OpenMP/MPI program was made malleable by exploiting MPI-2’s process creation API. PVM’s pvm_spawn and pvm_kill implement the same functionality by spawning and terminating PVM processes in the application. Both MPI-2 and PVM rely on the programmer to add and remove SUEs as necessary, as an automatic solution is impossible (see Section 4.3); JaMP handles this for OpenMP-parallel applications automatically.

As changing the number of UEs that execute the program, comprises a data and work redistribution, a runtime for malleable applications needs to deduce the distribution schemes implemented by the programmer. Several approaches exist that provide APIs for the programmer to inform the runtime about the distribution scheme. Others avoid the problem by keeping the
work distribution for the UEs fixed and introduce SUEs to actually execute
the program. In the following, we present some of these approaches.

GrADS applications [201] may alter the degree of parallelism after a migra-
tion to a target system. Only iterative MPI applications that are explicitly
designed by the programmer for malleability are supported. The MPI ex-
tension of [133] exhibits an API to inform the MPI library about how to
redistribute data and work if the number of processes is changed. GrADS
and [133] force the programmers to learn and accept a new programming
paradigm. With our malleability solution, they can re-use existing OpenMP
knowledge and leave the task of reparallelization to the compiler and its
runtime system.

The MPI programs of [69] and AMPI [95] do not perform true reparalleliza-
tion to implement malleability. Instead, the application is over-decomposed
(see Section 4.2.1) for a high number of virtual MPI processors that are
in turn mapped to physically existing MPI processes by multiplexing tech-
niques (see Section 3.5). If a virtual process enters an MPI primitive, the
runtime system schedules another MPI process if necessary. As will be dis-
cussed in Section 4.2.1, this approach suffers from limited scalability, since
the application cannot utilize more processing elements than the number of
virtual processors it started with. Our approach avoids these drawbacks.

Cactus [76] and DGET [63] adapt to changes in the computing environment
and can acquire or release nodes while an application is running. Although
both fulfill our requirements for malleable applications, Cactus and DGET
enforce own programming models in which the application has to extend a
framework with callbacks to application-specific functionality. This breaks
with the requirement of transparency for the programmer. Our solution does
not require the OpenMP application to necessarily extend a framework or to
use special APIs. Our approach is less invasive, since unmodified OpenMP
applications can benefit from malleability.

Considering malleability in OpenMP applications, our system is the only
available approach to change the degree of an OpenMP application while a
parallel region is active. Although implementations adhering the OpenMP
specification may alter the number of threads that execute a parallel re-

gion [154], in existing implementations except ours, the thread number is
kept fixed during execution of parallel regions. If an OpenMP application
only contains a single, long-running parallel region, the application effect-
ively becomes moldable and may not change its degree of parallelism.
The OpenMP extension presented in [191] offers deferred cancellation features for OpenMP applications processing irregular data structures. Threads can be scheduled to later exit at certain cancellation points in the parallel region; programs cannot create new threads while a parallel region is active. This chapter presents a reparallelization solution that adapts to the available computing resources by both adding and removing threads.

The OpenMP implementation [131] that runs on a Network of Workstations (NOW) supports malleable OpenMP programs to free or claim nodes in the NOW. If a workstation becomes idle it joins the DSM used for the OpenMP implementation. However, no additional OpenMP thread is spawned to join the currently active parallel region. Instead, the new workstation is left idle until the program re-enters the sequential code. If later on a new parallel region is encountered, the idle workstation is incorporated by spawning new threads. If a workstation has to be dropped because of increased user load, the threads running on this machine are migrated to another workstation. This approach basically implements inter-phase alteration, as the number of threads is kept fixed during parallel execution. Hence, JaMP is more flexible and may adapt the application’s parallelism to the computing environment while parallel regions are active.

4.2 Malleability in OpenMP Applications

Figure 1.5 has shown that approaches to make an application malleable can roughly be classified into four categories: inter-phase alteration, multiplexing, reparallelization, and hybrid approaches. As we are considering OpenMP applications in this work, inter-phase alteration can be ignored, as it is already defined by the OpenMP specification (see `omp_set_dynamic` and `OMP_DYNAMIC` in [154]).

The rest of this section focuses on the other approaches to implement malleability for applications that alter their degree of parallelism while they are running: multiplexing, reparallelization, and hybrid approaches mixing both concepts.
4 Reparallelization of OpenMP Constructs

### 4.2.1 Multiplexing

The problem of making an application malleable can be solved by creating a large number of LUEs that execute the application’s parallel regions. These LUEs are then multiplexed on a smaller set of SUEs that actually reside in the execution environment. In the literature, this approach is also called *over-decomposition*, as the problem that is solved by the application is overly decomposed to more LUEs than PEs that are actually available in the executing environment. Similar approaches are used in multi-processing environments; in such environments, the operating system schedules a potentially large number of concurrently executed processes on the available CPUs of the machine [173, 192].

In Chapter 3, we have shown a similar approach that multiplexes several logical OpenMP threads on the system units of execution. The scheduler stores the computational state of a postponed LUE in a continuation and replaces the LUE with the state of another LUE. Similar approaches are used in prior works to schedule several logical MPI processes on physical MPI processes [69, 95].

As the number of LUEs and SUEs is decoupled, the number of SUEs can be changed without affecting the number of logical UEs. If new processing elements join the computation, the scheduler adapts the number of system UEs accordingly and alters the scheduling policy such that the new SUEs receive LUEs for execution. If PEs and their corresponding SUEs are removed, the logical UEs are moved from the removed SUEs to the still existing ones.

Using over-decomposition has two major disadvantages. First, the number of logical UEs is determined when the application is started for the first time. After start-up the number remains fixed until the application has completed computation; only the number of SUEs is changed during execution. As a matter of fact, the application’s maximum degree of parallelism is limited by the number of logical UEs, even if the application’s scalability allowed for a higher degree of parallelism. If the number of SUEs is increased, no additional speed-up can be achieved. If the number of SUEs is equal to the number LUEs, each SUE is assigned at most one LUE. If increased even further, the additional SUEs will remain idle, as they will not receive any logical UE. Thus, the maximum number of logical UEs has to be chosen carefully such that it matches the expected highest number of PEs in the execution environment.
4.2 Malleability in OpenMP Applications

Second, as each LUE is subject to scheduling on an SUE, the LUEs need to be postponed when another logical UE is scheduled to run on a system UE. As we have seen in Section 3.5.1, the complete computational state has to be stored for postponed LUEs. Hence, increasing the number of logical UEs also consumes memory that is not available to the application. As a consequence, the number of LUEs has to be limited such that the application does not suffer from memory starvation.

4.2.2 Reparallelization

Multiplexing does not change an application’s parallelism at the program level, but alters the PE count by adapting the SUE scheduling policy for LUEs. In contrast, reparallelization [115] modifies the actual degree of parallelism by changing the number of LUEs (i.e., OpenMP threads) that execute the currently active OpenMP parallel region of the application. The work and data distribution of the application has to be changed to reflect the new number of PEs in the execution environment.

If new processing elements become available, the data and work distribution has to be modified such that the additional PEs are fully utilized by the LUEs of the application to achieve the best level of performance. Adding new LUEs to the application implies a redistribution of the work assigned to the already existing LUEs. Each processing element should receive roughly the same amount of computational load to avoid a load imbalance. If processing elements are removed, the excess LUEs have to be merged to avoid overloading the processing elements. This again requires a redistribution to move work from the removed UEs to the remaining ones. Reparallelization can only be performed if the complete information about the parallelization and the data partitioning of the parallel region is available to the reparallelization system. If it is not able to deduce the information about parallelization, it cannot adjust the work and data distribution.

To unveil the information about the parallelization of an application to a compiler, three options arise. First, a compiler can automatically parallelize the sequential code of the application [19, 84, 179]. As the compiler transforms the application, it can deduce the data and work distribution from the code templates used to perform the parallelization. Automatic parallelization in compilers is a difficult problem and limited to a certain class of applications [98].
Second, in semi-automatic approaches to parallelization, the user augments the sequential code with parallelization hints that are interpreted by the compiler and used to transform the code into a parallel program. Examples of such languages are High Performance Fortran (HPF) [120], Unified Parallel C (UPC) [199], and OpenMP. Such languages alleviate the compiler from the task of analyzing the application for parallelizable regions. The compiler only has to apply code transformations that emit parallel code. This again enables the compiler to deduce the data and work distribution from its transformation templates.

Third, in manually parallelized codes, the programmer rewrites the sequential code to create a parallel version. Since a programmer has unlimited degrees of freedom to write parallel codes, it is virtually impossible to infer the distribution scheme, as Section 4.3 will show.

Since we focus on OpenMP applications in this work, we restrict ourselves to semi-automatic approaches to perform a parallelization of applications. For each parallel region of the application, the compiler applies the transformation templates shown in Chapter 3 to emit parallel code. Hence, the parallelization scheme applied to transform the application code into parallel code is known to the OpenMP compiler. Deriving the distribution scheme from the code transformation template, the compiler can add code to the application that performs the reparallelization and adapts the data and work distribution accordingly.

4.2.3 Hybrid Approach

Multiplexing and reparallelization may be combined into a hybrid approach to make parallel applications malleable. As multiplexing creates an overly large number of LUEs and schedules them on SUEs, increasing the number of SUEs eventually yields a 1:1 mapping of LUEs to SUEs. As we have seen, this limits the application’s scalability.

A hybrid approach overcomes this limitation by starting the application with a certain number of LUEs. If the number of SUEs is about to reach the number of LUEs, the reparallelization runtime is triggered to increase the number of LUEs such that new SUEs can be added when desired. As the number of LUEs is always greater than the SUE count, the application’s scalability is no longer limited by the initial LUE count. A different hybrid
4.3 Parallelization Analysis with Data-flow Analysis

Let us further investigate the problem of deducing the work distribution of a manually parallelized program using the example code from Figure 4.1 and Figure 4.2. The code is inspired by the earlier OpenMP example of Figure 2.2 and computes the sum of the elements of an array. Figure 4.1 shows the main program that spawns the threads and partitions the work; Figure 4.2 depicts the definition of the threads and the reduction. The program is a simple example showing a straightforward parallelization of this trivial algorithm.

As can be seen from Figure 4.1 (lines 18–23), each thread receives a partition of the array to work on. Using the ID of the thread, the code computes the upper and lower bound of the array partition. The thread is then started to compute its partial sum on its array partition (see the loop in Figure 4.2, line 29). After the loop, the threads contribute the intermediate results to the global result in line 32 of Figure 4.2. For simplicity, we assume that the size of the array is a multiple of the number of threads. The loop in line 25 of Figure 4.1 uses Java’s thread synchronization primitives to block the main thread until all threads have completed their work and terminate. Finally, the result is retrieved from the reduction object.

When a reparallelization occurs, some fields of SumThread have to be altered to adapt the data/work partitioning to the new number of LUEs, as each thread privately stores its boundaries of its data/work partition in fields. In this example, the compiler’s data flow analysis [81, 179] has to analyze the main loop in the run method (line 29, Figure 4.2) of the thread class to detect that the fields low and high of SumThread determine the data partition that a thread works on at runtime. Using interprocedural DFA, the compiler has to trace back to the definitions of low and high in the constructor of SumThread (lines 22 and 23). Hence, it can determine the
class Main {
    static final int DEG = 4;

    public static void main(String[] args) throws InterruptedException {
        int len = 1024;
        int arr[] = new int[len];

        int compare = 0;
        for (int i = 0; i < len; i++) {
            arr[i] = i * i;
            compare += arr[i];
        }

        Reducer red = new Reducer();
        Thread[] threads = new Thread[DEG];
        for (int id = 0; id < DEG; id++) {
            threads[id] = new SumThread(arr, len / DEG * id, len / DEG * (id + 1), red);
            threads[id].start();
        }

        for (int id = 0; id < DEG; id++) {
            threads[id].join();
        }
        int sum = red.getResult();
    }
}

Figure 4.1: Example of a manual parallelization of the array sum example (main program).
class Reducer {
    private int res = 0;

    synchronized void add(int sum) {
        res += sum;
    }

    synchronized int getResult() {
        return res;
    }
}

class SumThread extends Thread {
    private int[] arr;
    private int low;
    private int high;
    private Reducer red;

    SumThread(int[] arr, int low, int high, Reducer red) {
        this.arr = arr;
        this.low = low;
        this.high = high;
        this.red = red;
    }

    public void run() {
        int sum = 0;
        for (int i = low; i < high; i++) {
            sum += arr[i];
        }
        red.add(sum);
    }
}

Figure 4.2: Example of a manual parallelization of the array sum example (thread and reducer).
flow of values through the two fields from their usage to their definition during the construction of the thread object.

After the data flow in the \texttt{SumThread} class has been analyzed, the compiler needs to track all object creations that construct a \texttt{SumThread} instance. From the constructor calls, the compiler has to deduce the expressions that are evaluated to bind the formal parameters of the constructor call. As these expressions may contain variables, the DFA has to recursively trace back to the definitions of the variables. If the expressions contain method calls, the DFA has to enter the body of these methods to perform an interprocedural analysis to gather information about the values the methods may return.

From the above description, it is obvious that it is a complex task for the compiler to actually deduce the data flow that constitutes the data and work distribution of a parallel program.

In the general case this complex task becomes practically impossible, as the programmer has too many degrees of freedom to express the computation of the data and work distribution. For example, a different approach for the example of Figure 4.1 and Figure 4.2 could compute the partitioning by passing the thread ID to the threads' constructor and calculating the boundaries of the work partition in the threads' \texttt{run} method. The programmer could also splice-out the computation into a distribution function that assigns partitions to threads. As many solutions to the partitioning problem exist, it becomes virtually impossible for a compiler to automatically deduce a programmer-written data and work partitioning. Generally, it is the halting problem \cite{198} that limits the compiler's ability to infer the exact semantics program code.

\section*{4.4 Reparallelization of OpenMP Work-sharing Constructs}

Most OpenMP applications process data structures such as arrays or matrices in parallel. The OpenMP work-sharing constructs parallelize \texttt{for} loops that process such data structures by partitioning the iteration space. The partitions are assigned to the threads of the thread team (see Chapter 2) to distribute the total work of the loop among the threads.

It is crucial for a reparallelization approach to provide effective means to change the number of threads during the execution of work-sharing con-
4.4 Reparallelization of OpenMP Work-sharing Constructs

The reparallelization of OpenMP-parallel loops must maintain correct semantics of the loops as otherwise the program’s correctness would be violated if a reparallelization was performed. Section 4.4.1 analyzes central properties of loops in general and in particular for OpenMP-parallel loops. We then introduce the notion of adjustment points at which a reparallelization of OpenMP constructs may happen (Section 4.4.2). We argue why such adjustment points are needed and how they might affect program execution. Finally, we introduce partial chunk descriptors that save the execution state of loops (Section 4.4.3) and cover reparallelization for the different OpenMP loop scheduling types (Section 4.4.4).

4.4.1 Anatomy of a Loop

Each for loop of a program is governed by a central and basic loop invariant [54, 90] that ensures that each iteration of the loop is executed exactly once. If this invariant was violated, the program’s semantics would be changed, as results that are computed by an iteration would be computed twice or may not be computed at all. Repeated or lost side-effects or accumulation of partial results would change the program’s states visible to the programmer. It is obvious that the invariant has to be respected both in sequential and parallel executions of a loop. In addition, keeping the invariant is a key requirement for reparallelization of parallel loops. If threads are removed, uncomputed iterations that were assigned to these threads must be computed later on. If threads are added to the execution, already computed iterations must not be computed again. Hence, the exact shape of the iteration space of a loop must be known for a reparallelization.

Informally, the iteration space describes which values of the individual loop counters become valid during the execution of the loop nest. In general, it is a difficult and complex task for a compiler to completely analyze the iteration space of arbitrarily nested loops [13, 21]. The loop variables of n nested loops span an n-dimensional vector space, in which each valid setting of the loop variables constitutes an integer vector. To compute the iteration space, the compiler has to derive the iteration vectors from the initialization expression of the loop variables, the loop boundaries, and the loop’s increment expression.

This complex task of analyzing loop nests can be omitted in the case of OpenMP. The OpenMP specification requires that only a single loop of a loop

114
nest can be parallelized and the shape and size of a parallel loop’s iteration space is known before the loop is entered (see Section 2.3.3). Hence, the work distribution can be computed before the loop execution starts. Before the parallel execution of a loop, its iteration space is distributed such that each thread receives a disjoint part of the iteration space of the loop. Depending on the scheduling type, each thread receives exactly one chunk of the loop (static) or executes several chunks of potentially different sizes (static with block size, dynamic, and guided). It should be immediately obvious that, independent of the scheduling type, each iteration of the parallel loop is executed exactly once.

Figure 4.3 shows the iteration space of a statically scheduled loop. Each single iteration is depicted as a block in the figure; the loop is cut into equal-sized chunks (indicated by thick vertical lines). Threads 1 and 2 already started computing their loop chunks and finished execution of the first two iterations of their respective chunks (marked “X”). Furthermore, thread 2 is currently working on the third iteration of its chunk and already processed parts of it. Threads 3 and 4 did not start processing yet.

If a thread is removed from the computation, its chunk has to be computed later on to ensure that each iteration is executed exactly once. The postponed chunk may contain processed iterations, partially finished iterations, and uncomputed iterations (see Figure 4.3). Partial chunks emerge, when a thread is removed after it has finished the current iteration of its chunk and before it enters the next one. Partial iterations originate from removing a thread at an adjustment point inside the loop body.
4.4 Reparallelization of OpenMP Work-sharing Constructs

The task of reparallelization of a parallel loop implies a work redistribution. Hence, the OpenMP compiler has to emit code that performs the redistribution of work when the number of threads is changed. However, adding the complete reparallelization code to the loop body would lead to an undesirable growth of the code size and would increase the pressure on the instruction cache of the CPU. To avoid these problems, the compiler does not insert the actual reparallelization code, but leaves a small stub in the loop body that triggers external reparallelization code in the runtime system. The reparallelization stub is an additional if statement that tests whether or not a reparallelization is requested and that triggers the appropriate runtime functions (see Section 4.7).

It is desirable to allow for reparallelization to occur on a very fine-grained level, that is, at (virtually) every machine instruction of the loop body. For obvious reasons, this is impossible as it would imply to add the reparallelization stub to each machine instruction. Although this approach would provide the finest granularity of reparallelization, it would deteriorate performance because of the overly large overhead caused by the huge number of stub executions. In addition, too many executions of the reparallelization stubs again pollute the instruction cache and further decrease performance.

The only viable solution is to restrict reparallelization to certain points in the program, so-called adjustment points. We define the new OpenMP directive \#omp adjust, which can be inserted either manually by the programmer or automatically by the compiler (see below). An example of the adjust directive is shown in Figure 4.4. The format of the directive follows the well-known syntax of other OpenMP directives such as the barrier or the flush directive. Threads can enter the parallel region or existing threads
can be removed from the region only at adjustment points. Between two consecutive adjustment points in an OpenMP-parallel region, the number of threads remains unchanged.

The placement of adjustment points is restricted by the following rule (a similar restriction applies to the barrier construct). As an adjustment point denotes a global operation that involves all threads executing a parallel region, it has to be encountered by all executing threads or by no thread at all. Hence, it is disallowed to place an adjustment point in the conditional branches of if or switch statements. In this case, the compiler aborts compilation with an error message for manually placed adjustment points.

Our research prototype also provides a simple algorithm that automatically inserts adjustment points in parallel regions and work-sharing constructs. Adjustment points should be distributed evenly over a parallel region such that a reparallelization can happen as early as possible. The following algorithm places adjustment points such that they are evenly distributed and encountered by all threads. In addition, the number of live variables at the adjustment point is kept as small as possible (see the next section).

In a preparation phase, the algorithm first constructs the Control Flow Graph (CFG) of the parallel region or the work-sharing construct for which adjustment points shall be added. In the CFG, the compiler assigns a fractional number $f (f = 1$ in a function’s entry vertex) to each vertex of the CFG. If a vertex has more than one successor, $f$ is divided by the number of successors and the resulting fractions are assigned to the successors. If paths of the control flow graph merge in a vertex, the fractions of each vertex are added and assigned to the merging vertex. If the fraction $f$ assigned to a vertex equals one, the vertex is not inside a conditional branch and is thus reached by all threads.

After the preparation phase, the algorithm inspects each instruction in a CFG vertex with $f = 1$ of the region and assigns a weight to it. The weight for regular instructions (e.g. arithmetic instructions, load/store instructions, etc.), is low whereas the weight for call instructions that transfer control to a function is high. During inspection of the code, the weights are added to a counter $c$. If the counter reaches a given threshold, an adjustment point is inserted and the counter is reset to zero. To avoid performance problems in loop nests because of introducing too high overheads caused by added adjustment points, the compiler only inserts adjustment points to the outermost loop level.
4.4 Reparallelization of OpenMP Work-sharing Constructs

4.4.3 Partial Chunk Descriptors

A dynamic reparallelization must take into account which fraction of the iteration space has already been computed and what remains to be done by the changed number of threads (see Figure 4.3). For maximum flexibility, adjustment points may be placed at arbitrary code locations in parallel regions and work-sharing constructs. Hence, at a given point in time, some chunks of a loop are still unprocessed, others are partially processed (i.e., some of their iterations are finished), and some chunks are already completely processed. In a partially completed chunk, some iterations have been computed, while other iterations still need to be executed. There might even be partial iterations, that is, iterations whose execution has been started but not finished yet.

If threads are removed at an adjustment point in the body of a work-sharing construct, other threads have to take over the remaining uncomputed work of the removed threads. We again adapt the idea of continuations to store the computational state of the loop. A partial chunk descriptor records the computational state and is stored in a set of uncompleted chunks for later resumption. The descriptor stores an iteration space description of the thread’s partition, that is, the current value of the loop counter, the loop’s lower and upper bounds, and the step width. In more complex situations (see below) the descriptor additionally contains a bit vector. To resume the chunk’s postponed execution, the partial chunk descriptor has to memorize the ID of the adjustment point that caused the creation of the descriptor.

To uniquely identify the adjustment points of a parallel region we have to use an ID, as the machine address of the executable code of the adjustment point is not suitable. As we target clusters with our solution, no global address space exists. Hence, an address that is valid on one machine is not necessarily valid in the address space of another machine. Using an artificial identifier solves this problem, as each machine can provide the corresponding mapping of the ID to a machine address in its local address space. We will come back to the mapping later, when we present the code template for the adjustment points.

Since the computational state of a loop chunk also contains any local variables that are live at the adjustment point, the partial chunk descriptor has to store a copy of the live variables. We rely on standard compiler analysis to determine the set of live variables [16, 81]. The types of the variables have
to be known to the compiler to be able to generate the code that copies the
values of the variables into the descriptor object. Thus, our reparallelization
runtime requires a type-safe environment, which we can assume in case of
Java and Fortran as well as for most C/C++ programs.

When a new thread joins the thread team, it can grab any unprocessed chunk
and start executing the region’s code for that particular chunk. If no un-
processed chunks are left, new threads have to take over partial chunks that
have been postponed by formerly removed threads. But, instead of starting
execution of the code of the work-sharing construct from its beginning, the
chunk has to be resumed at the code location it was stopped at. As already
mentioned, the adjustment point is identified by a unique number and the
mapping of the ID to the code location is implemented by means of a branch
table. Similar branch tables are, for example, created when a compiler emits
code for switch statements with dense selector values [66, 168]. The unique
index is used to retrieve the address of the branch target and an indirect
branch instruction continues execution at the target address.

4.4.4 Loop Schedule Types

OpenMP defines different types of loop scheduling clauses that a programmer
can choose from to distribute loops among the threads that execute the
OpenMP work-sharing construct. The valid OpenMP loop scheduling types
are static, static with chunk size, dynamic, and guided (see Section 2.3.3).
As each scheduling type distributes the loop differently, reparallelization has
to respect the different scheduling types and has to redistribute the loop’s
work accordingly.

Static scheduling with explicit chunk size

A static loop schedule with an explicit chunk size divides the iteration space
of a loop into equally-sized chunks. The chunk size is determined by the
schedule clause at the OpenMP construct; only the very last chunk of the
loop may be smaller if the loop’s number of iterations is not a multiple of the
chunk size. The chunks are then assigned to the threads in a round-robin
fashion.
4.4 Reparallelization of OpenMP Work-sharing Constructs

This assignment has to be updated when the number of threads in the team changes. Since a new assignment can only be computed if it is known which chunks of the loop have already been processed, every thread memorizes the list of completed chunks. If the number of threads changes, the list is used to redistribute the uncompleted chunks of the loop and to assign each thread a new set of chunks to work on. However, for loops with a large number of chunks the process of redistributing the chunks may cause severe performance penalties. In addition, the reassignment algorithm has to be stable, that is, after the reassignment the moved chunks should be as close as possible to their formerly assigned threads. We adopt the term “stable” from sorting algorithms: a sorting algorithm is said to be stable if it does not change the relative ordering of the equivalent elements during sorting [92].

The reassignment is performed in two phases. First, the regular assignment is repeated for the new thread count assigning all chunks to the new set of threads in a round-robin fashion. At this stage, we do not distinguish unprocessed and processed chunks, as the completed chunks can be skipped later on when the reparallelization is finished and the threads resume their normal operation. A possible result of this algorithm, when adding two threads to two already existing threads, is depicted in Figure 4.5(a). Each thread received six equally-sized chunks during the initial assignment when the OpenMP construct was entered. At the time of the reparallelization, thread 1 already completed four chunks, while thread 0 started to execute but did not finish any chunk. After the first phase, the chunks are distributed in round-robin fashion. Each of the four threads receives the same number of loop chunks, but thread 1 and thread 3 receive less work, because thread 3 mostly receives finished chunks from the former thread 1.

Second, the assignment computed by the first phase is improved to reduce the load imbalance. The second phase relocates chunks from overloaded threads to undersupplied ones. We adapted a greedy algorithm from [6] for this purpose. The algorithm compares the number of processed and unprocessed chunks for each pair of neighboring threads. If a thread was assigned more chunks than its neighbor, the algorithm moves half of the excess chunks from the overloaded thread to the undersupplied ones. This process is repeated until the chunk distribution is balanced over all threads. Figure 4.5(b) depicts the process of relocating chunks for the example distribution of Figure 4.5(a). As thread 0 and thread 2 are overloaded, one chunk is moved to thread 1 and thread 3, respectively.
Figure 4.5: Reassignment example for a static loop schedule.
4.4 Reparallelization of OpenMP Work-sharing Constructs

This algorithm does not dynamically balance the load on the threads. It just performs a static relocation, as it only uses the number of chunks to determine the load and ignores computational load that stems from the execution of individual chunks. This conforms to the requirements set up by the OpenMP specification for static scheduling on loops. In addition, the algorithm tries to respect locality issues, as chunks are moved between neighboring threads where possible to avoid scattering work over the PEs.

**Static scheduling without explicit chunk size**

If no loop scheduling type is specified or a `static` clause without an explicit chunk size is added to the work-sharing construct, the iteration space is divided such that each thread receives exactly one chunk. Hence, a redistribution scheme for static loop scheduling as described above is not applicable in this case. As there are only as many chunks as there are threads, new chunks need to be created for joining threads. Since each thread already processed parts of its chunk, the iteration space is additionally fragmented into disjoint sequences of processed iterations and unprocessed iterations.

As we need to memorize the state (i.e., processed or unprocessed) of individual iterations, an efficient data structure has to be used. The data structure must not consume too much memory for loops with many iterations. In addition, storing the state of an iteration has to be efficient, as the state is updated after each iteration of the loop. We use a bit vector for this purpose. Each iteration is assigned a bit in the vector. If the bit is not set, the iteration is still uncomputed. If set, the iteration was already computed and finished by one of the threads. While providing a compact storage of iteration states, setting a bit in the vector only adds a few instructions to the instruction stream of an iteration. Partial iterations cannot be reflected by a single bit and are thus stored in an additional list-like structure.

For each iteration, the base address of the vector has to be loaded. Then, the loop counter is divided by 32 to obtain the address of the word storing the iteration’s bit. The remainder of the division is used as index into the word and the bit is flipped. The instruction set of the hardware provides corresponding low-level instructions to efficiently access the bit vector. Synchronization can be omitted, as the bit vector is a thread-local vector that is concatenated to a global bit vector when reparallelization is performed.
During reparallelization, the runtime system inspects the bit vector and counts unprocessed iterations. Similar to the case of static scheduling with chunk size, each thread receives roughly the same number of unprocessed iterations. Hence, after reparallelization, the number of chunks is again equal to the number of threads. However, each chunk may contain processed iterations, which have to be skipped by the threads during execution of their respective chunk. This also involves a bit test that can be implemented efficiently as well, by replacing the bit set operation of the above code sequence with a bit test and a branch instruction to skip the loop body if the bit is set.

**Dynamic and guided scheduling**

With dynamic loop scheduling and guided loop scheduling threads request small chunks from a (conceptual) global work pile and no static mapping of the iteration space to threads exists. This can be seen as a special and very limited case of the \( m:n \) mapping of LUEs to SUEs that was presented in Section 3.5.1. The iteration space of the loop is over-decomposed into a number of chunks. Each chunk of the loop can be considered as an LUE that has to be dynamically scheduled to run on an SUE (the OpenMP thread). The scheduling is not performed preemptively but cooperatively, as each thread requests chunks from the work pile when it finishes computing its current chunk.

New threads can participate in the computation without the need to change the number of chunks or redistribute chunks that have been assigned to the threads for execution. A joining thread enters the OpenMP construct and requests a chunk from the work pile for processing. If there are unprocessed chunks or partial chunks left in the work pile, they are automatically assigned to the new threads. If the global work pile is empty, the new threads remain idle. Changing the number of chunks in this case is not allowed, as the chunk size in a dynamic loop schedule is prescribed by the programmer and may not be changed. In case of a guided schedule, the runtime system has to respect the new number of threads, when the chunk size is recomputed as requested by the OpenMP specification.
4.5 Reparallelization of Other OpenMP Features

Reparallelization of OpenMP-parallel loops is not enough to cover all possible parallelization issues of OpenMP programs. We still need to consider altering the degree of parallelism for other OpenMP constructs such as parallel sections and for arbitrary parallel regions. Finally, reductions need to be handled, as they play an important role for almost all kinds of parallel regions in OpenMP programs.

4.5.1 Parallel Sections

Using parallel sections, the OpenMP programmer can write programs in which each thread executes a different part of the application concurrently (to the other threads of the team). Each subsequent section contained in the `parallel sections` construct is assigned to a thread for execution. As Section 3.3.4 has shown, this is accomplished by transforming a `parallel sections` construct into a `parallel for` construct that contains a single `switch` statement with one `case` block for each `section`. Distribution of the different `sections` among the threads is achieved by using a `static` loop schedule with chunk size 1. With this transformation, each loop iteration corresponds to exactly one of the `section` constructs of the `parallel sections` construct.

As `parallel sections` constructs are replaced by a regular work-sharing construct, they can be automatically handled by the reparallelization approach for work-sharing constructs. If a reparallelization occurs, the parallel loop of the work-sharing construct is redistributed such that the number of threads receive roughly the same amount of work. As the different sections of the `parallel sections` construct correspond to a single iteration, they are automatically distributed among the new number of threads in the team.

4.5.2 Parallel Regions

Work-sharing constructs partition the iteration space among the threads of the team by assigning a partition of the iteration space to each thread. To maintain correct loop semantics, the total work has to remain the same even if a reparallelization of the work-sharing construct is performed. In contrast
to work-sharing constructs, the semantics of a parallel region in OpenMP prescribe that the same code is run by all threads that execute a particular parallel region. Hence, the total amount of work performed is determined by the number of threads that execute the parallel region: doubling the number of threads also doubles the amount of work performed.

This observation is crucial for the reparallelization of parallel regions, as removing threads and adding threads has to alter the amount of work performed by the new number of threads. If threads have to be removed at an adjustment point inside a parallel region, the thread is terminated by the runtime system and its computation is aborted. Hence, after the adjustment point, only the remaining threads continue to execute the parallel region. If threads are added, they join execution at the adjustment point and begin executing the code after the adjustment point. In doing so, the semantics of a parallel region is directly reflected in the reparallelization, as the total amount of work performed at any point in the parallel region is always proportional to the number of threads currently active.

In Section 4.7, we will return to this and present details on how threads can be added to a parallel region.

### 4.5.3 Reductions

Reductions are used to accumulate partial results of threads into a single, global result at the end of a parallel region. Each thread computes a partial result for its partition of the work-sharing construct or for its computation performed during a parallel region. After the parallel region is finished, these results are aggregated to the global result. Obviously, reparallelization has to respect the semantics of reductions to avoid breaking the correctness of the program.

Let us revisit the details of reductions. In a work-sharing construct, every single iteration of the parallel loop may contribute to the global result. However, the global result is spread over the partial results of the threads until the reduction was eventually performed at the end of the parallel region. At the beginning of the parallel execution, the value of the partial result for each thread is set to the neutral element with respect to the reduction operator. For example, the neutral element for reductions with the “+” operator is the
value 0. During computation, each thread contributes values to its partial result by continuously applying the reduction operator.

If a work-sharing construct is reparallelized and threads are removed, the removed threads leave behind uncomputed chunks of work. These chunks have to be finished before the work-sharing construct is exited. However, as each iteration of these chunks also contributes to the partial results of the removed thread, the partial result has to be saved with the uncomputed chunk. When the chunk is resumed later on, the saved result is reloaded and the remaining iterations are finished.

Similar to the initially created threads of a team, a thread created by a reparallelization has to initialize its partial result for a reduction, if it joins the computation of a work-sharing construct and receives uncomputed chunks of work. Two possible cases arise. First, if the joining thread receives an uncomputed chunk of the parallel loop, it can start executing the chunk. As the chunk is still uncomputed, it does not have a partial result associated with it. Hence, the thread initializes the partial result with the neutral element of the reduction operator and starts processing of the chunk. Second, the joining thread may receive a partial chunk descriptor for an unfinished partial chunk. In this case, the partial chunk descriptor stores a partial result from the formerly executing thread and the new thread has to merge its current partial result with the partial results that came along with the unfinished chunk.

Although this approach reorders the sequence of applications of the reduction operator, it remains a valid reduction with respect to the requirements of the OpenMP specification. OpenMP allows that the results of a reduction slightly differ, as it does not prescribe an ordering for the reduction of partial results.

### 4.5.4 Thread Contexts

When a partial chunk is assigned to a thread for execution, the partial chunk descriptor contains the ID of the adjustment point and information about the boundaries of the loop partition contained in the chunk. As it also stores all live variables, a partial chunk descriptor is able to store the computational state of a terminated thread’s loop chunk. The joining thread can load the necessary data from a loop chunk and start execution. Thus,
4 Reparallelization of OpenMP Constructs

no further actions are required for threads that are added to the execution by a reparallelization.

For added threads that do not receive a postponed chunk and that start at an adjustment point in an OpenMP construct, no partial chunk descriptor is available. Variables that are live at the adjustment point have to be initialized with valid values for the new threads. The value cannot be computed by the thread, as it joins computation at a code location that is located after the definition of the variables. To actually obtain the values for the initialization of such variables, two different approaches seem appropriate.

First, the runtime system could re-execute the computation that determined the values of all live variables. Alas, this solution is only applicable for computations that do not cause any side effects. Without side effects, redoing the computation becomes possible, since the result of the computation only depends on the input values and, thus, always gives the same result for the same set of input values. In functional programming, this is a fundamental concept known as referential transparency \([1, 81, 177]\). For Java, the compiler in general is not able to completely analyze the program for referential transparency. Due to the halting problem compilers can only provide a conservative approximation in the best case \([16]\).

As each method of the call graph rooted at a variable’s initialization expression might cause side effects, the compiler would have to analyze the call graph to determine all potential side effects \([13]\). At runtime, Java programs can use the class loader facility to load code that was not available during compilation \([126]\). It is therefore impossible for a compiler to actually determine the exact call graph of the program \([20, 51, 80, 86]\). As a consequence, the compiler is not allowed to insert code that repeats the computation to obtain the values of live variables, as the compiler is not able to guarantee correct program semantics.

Second, copying the values of live variables of other threads is an option. If a thread is added to the work force executing an OpenMP construct, it retrieves the values for live variables by copying them from another thread. To preserve correctness, the semantically correct value of the live variable has to be used. As each thread may change the value of a variable during the computation that follows the adjustment point, only values that were valid at the adjustment point may be copied to the newly created thread. To comply with the OpenMP specification, we use the master thread as the origin of values for live variables. When it reaches an adjustment point, the
4.5 Reparallelization of Other OpenMP Features

The master thread creates copies of its live variables at the current adjustment point and, thus, preserves the current values of the live variables. Later on, an added thread can retrieve the values and initialize its live variables with these values.

To store the values of variables, we have to distinguish primitive data types and object types. For variables that are of a primitive data type (e.g. `int`, `double`, etc.), the value can be assigned directly to the master thread’s copy. For variables with an object type (e.g. `Object`, `String`, etc.), the object’s `clone` method is used to create a copy of the object behind the reference contained in the live variable. In this case, the object’s type needs to implement the `Cloneable` interface. For other classes and if standard cloning does not produce a valid copy, special cloning facilities can be provided by the user. In Section 2.3.6, the `CustomCloner` interface was defined for this purpose. With calls to `ompAddCustomCloner`, a custom cloner used to create an object backup for the master thread can be specified (see Figure 2.6).

Although the second approach guarantees that live variables at an adjustment point are initialized with correct values, it has to conform to the semantics of the OpenMP data-sharing clauses. A `private` variable does not have to be considered here, as the value of the variable is private to the thread and is automatically initialized with the respective default value. For variables marked as `firstprivate`, the value from outside of the construct has to be used for the initialization of the private instance of a particular variable. As the master thread creates the backup of the value upon entering the construct, it automatically memorizes the correct value from the outside.

With the `shared` clause, each thread receives the same value of a particular variable, when entering an OpenMP construct. Updates to `shared` variables are visible to all other threads of the team as well. Hence, using the backup of the master thread for initialization of the thread context, newly created threads receive the same values as they would have received if they joined execution of OpenMP construct from the beginning.

Figure 4.6 illustrates the issues with thread contexts by giving an OpenMP example code. The example code defines a variable `c` of type `Collector` inside the parallel region, but outside the OpenMP `for` construct that contains the adjustment point. In addition, a variable `z` of type `int` is defined before the work-sharing construct. For simplicity, we omit the expressions used to initialize the variables. In the initialization expression, any valid, arbitrarily complex Java expression can be substituted for the “...” in the
code example. Both c and z are live at the adjustment point, as they are needed by the subsequent computation. When a new thread is added to the thread team, it has to receive valid values for both variables. When the master thread enters the work-sharing constructs, it creates copies of the values of c and z. A joining thread then receives these copies and initializes its instances of c and z.

### 4.5.5 Nested Parallelism

According to the OpenMP specification, it is possible to implement nested parallelism in an OpenMP application by recursively nesting parallel regions, that is, from within a parallel region a new parallel region may be created. It is up to the implementation how the threads for the nested parallel region are created by the runtime system. Some implementations create new threads and, thus, increase the number of threads, when a nested parallel region is entered. Others provide a fixed-size thread pool for the execution of all parallel regions. Threads for nested parallel regions are taken from the pool and used to execute the region. If no threads are left in the pool, the nested region is executed sequentially. In both cases, reparallelization blends well with nested regions.
4.5 Reparallelization of Other OpenMP Features

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class NestedParallelismExample {
    void foo() {
        // #omp parallel num_threads(4)
        {
            // region on level 1
            goo();
        }
    }
    void goo() {
        hoo();
    }
    void hoo() {
        zoo();
    }
    void zoo() {
        // #omp parallel num_threads(3)
        {
            // region on level 2
        }
    }
}
```

(a) OpenMP/Java code with nested parallelism spanning several methods.

(b) Execution of the nested parallel regions of the example in Figure 4.7(a).

Figure 4.7: Example for nested parallelism in OpenMP/Java.
Nesting of parallel regions may occur on different levels of the method call stack as shown by the example code in Figure 4.7(a). Figure 4.7(b) depicts the call stack and the created threads executing the parallel regions. In method \texttt{foo}, a parallel region on level 1 with four threads is executed. Each thread in \texttt{foo} calls a method \texttt{goo} that in turn calls \texttt{hoo}. Finally, \texttt{hoo} calls \texttt{zoo} that contains a nested parallel region on nesting level 2. As each individual thread from the top-level region in \texttt{foo} enters the parallel regions, the thread becomes the master thread and creates a new team.

When reparallelization is applied to the example execution of Figure 4.7(b), the reparallelization runtime can modify execution in two different ways. First, the number of threads can be changed on level 1 while the parallel regions on level 2 are actively executing. The thread context for added threads then not only consists of live variables, but also includes the call stack that led to the execution of a parallel region on level 2. Second, the runtime system can target level 2 during a reparallelization and keep the number of threads on level 1 as is.

From a programmer point of view, the second option provides a clean extension of the presented reparallelization features to nested parallelism. As the majority of the threads is active on level 2, nested parallel regions should be subject to reparallelization first. Therefore, the reparallelization runtime first targets the active parallel regions, activates the adjustment points therein, and alters the thread count on level 2. When execution unwinds the call stacks and resumes execution on level 1, the reparallelization runtime activates the adjustment points on level 1 and performs another reparallelization if needed.

### 4.6 Limitations

The previous sections have shown how the reparallelization of an OpenMP parallel region affects the semantics of an OpenMP application. It has also been shown that a compiler’s detailed understanding of an application’s parallelization is crucial to change the data and work distribution. Hence, there are limitations to the programming model if reparallelization is enabled. Another problem is that altering the thread count may cause data locality issues. Section 4.6.1 and Section 4.6.2 investigate these limitations and issues in detail.
4.6 Limitations

In Section 4.3, we have motivated the need for a data flow analysis of the programs with reparallelization capabilities. The compiler needs to gather detailed information about the parallelization that was performed by the programmer. Figure 4.1 and Figure 4.2 have shown a simple example that demonstrates the complexity of this task. Only programs that rely on OpenMP features for work distribution are suitable for reparallelization (so-called pure OpenMP programs).

4.6.1 Programming-model Limitations

```java
class Example {
    void dependent(double[] array) {
        // #omp parallel
        {
            int id = JampRuntime.getThreadNum();
            int cnt = JampRuntime.getNumThreads();
            int sz = array.length / cnt;
            int fr = sz * id;
            for (int i = fr; i < fr + sz; i++) {
                // computation using array[i]
            }
        }
    }
}
```

Figure 4.8: Dependency to the thread ID.

```java
class Example {
    void independent(double[] array) {
        // #omp parallel for
        for (int i = 0; i < array.length; i++) {
            // computation using array[i]
        }
    }
}
```

Figure 4.9: Corrected example of Figure 4.8.
4 Reparallelization of OpenMP Constructs

```java
class Example {
    void semantics() {
        // #omp parallel
        {
            System.out.println("A");
            // #omp barrier
            System.out.println("B");
        }
    }
}
```

Figure 4.10: Example code showing semantic issues with reparallelization.

Figure 4.8 shows an example of an OpenMP program that is not reparallelizable. The program uses OpenMP constructs to create the thread team for execution, but work distribution is programmed explicitly. While in this example, the OpenMP compiler could use DFA to analyze the work distribution, the example introduces an indirect data dependency from the thread ID to `array[i]`. Because of the data dependency, a reparallelization would cause undefined behavior, as a thread with a given ID might have been removed. Hence, if reparallelization is enabled, the compiler has to disallow the use of `getThreadNum()` and `getNumThreads()`. In the current implementation, the OpenMP/Java compiler emits a warning to indicate the use of these two functions to the programmer. However, this limitation is not severe, as the example of Figure 4.8 can be expressed with pure OpenMP constructs, as Figure 4.9 shows.

If threads are added to or removed from parallel regions, the number of executions of each statement changes. Obviously, this might affect program semantics as Figure 4.10 shows. If a thread is removed at the adjustment point, the program prints fewer “B”s than “A”s. If threads are added, the program prints more “B”s than “A”s. Because it may break program semantics, the programmer can disallow reparallelization by adding the `adjust(none)` clause to the `parallel` construct. The `adjust` clause is valid at every OpenMP construct and can be used to make any parallel region of the OpenMP rigid, that is, the number of threads is not altered during the execution of the parallel region. If `adjust(none)` is present at a clause, the compiler does not emit adjustment points to the construct and the use of manually placed `adjust` directives is prohibited by the compiler.
4.6 Limitations

To add threads to a parallel region at its beginning, the live variables of the master thread are copied to properly initialize live variables. However, this might alter the program semantics desired by the programmer. For example, a thread might store the current thread ID or the current number of active threads executing the parallel region, the newly added thread receives wrong values. In case of the thread ID, the added thread’s variable contains the ID of the master thread. A variable that saves the current number of threads contains a value that is no longer equal to the number of currently active threads after threads are added or removed. The programmer has to be aware of this and deal with it accordingly. However, since in most OpenMP applications the use of thread IDs and/or the number of threads indicates a weak application design (see Figure 4.8), this restriction is considered acceptable for the OpenMP with reparallelization features. Again, the compiler emits a diagnostic message that informs the programmer about the problem.

4.6.2 Data Locality Issues

As the number of threads is altered during the reparallelization, the assignment of threads to PEs is altered as well. On systems with a flat memory model (e.g. non-NUMA SMPs) this does not have an effect on program execution, except that the CPU caches have to be invalidated, which is automatically performed by the hardware. On systems with a NUMA architecture, changing the thread/PE assignment might cause a severe performance loss, as the application’s data layout might not fit the thread/PE layout anymore. Let us investigate this problem in more detail, since data locality is crucial to an application’s performance.

While providing a single address space that spans all PEs, the memory banks of a NUMA system are distributed among the system and are attached to the individual processors. If an access to the memory attached to another processor is needed, the accessing processor has to cross an interconnection network to reach the remote memory location and request the data from there [182]. This transition between local memory and remote memory is observed as an increased latency for memory accesses [93]. Examples of such systems are the SGI Altix, multi-processor systems based on the AMD Opteron, and the upcoming Intel Tukwila platform.
The memory architecture of S-DSM systems looks different. While S-DSM systems generally exhibit the same latency characteristics if the local memory is left and remote memory is accessed, they follow a different approach on making the remote memory available to the local PE. Being a Cache Only Memory Architecture (COMA) [105, 182], a DSM system accesses the remote memory by replicating the accessed page or object on the accessing processor. All accesses are then directed to this copy. Flush operations at memory fences cause the DSM system to write back the replica on the local PE to the remote memory location.

As in for both NUMA and COMA systems accesses to remote memories are more expensive than local accesses, it is desirable to direct memory accesses of the individual OpenMP threads to the local memory as often as possible [42]. Current implementations of operating systems for NUMA systems support applications by using a first-touch policy [26] for memory allocation. Memory pages are placed near the processor that first touches (i.e., reads or writes) a particular memory page. If a thread stays on the processor on which the thread has performed the allocation of data, future accesses to the allocated data remain local. In OpenMP applications, this is traditionally exploited by using work-sharing constructs for the data allocation as well as for the computation. However, performance of the computation depends on a fixed 1:1 mapping between the allocating threads and the working threads; this 1:1 mapping has to be maintained by the OpenMP runtime system by, for instance, pinning threads to CPUs.

Obviously, reparallelization breaks with the 1:1 mapping between allocating threads and working threads. If threads are added to or removed from the work-sharing construct constituting the main computation, it is no longer guaranteed that threads accessing data are still running on the same processing element that was earlier used for allocation of data. This may cause a severe drop of performance, as potentially each access to data in memory may be a remote access targeting memory areas of a different processor. There are two possible solutions to this problem: data migration and data redistribution.

With data migration, the memory accesses to remote memory locations are tracked. If it seems beneficial, the data is moved from the remote processor to the local processor. There are operating systems (e.g. current versions of the Linux kernel) that are capable of migrating pages between different PEs of the system to increase locality [205]. DSM systems also exploit this
4.7 Compilation of Adjustment Points

For each adjustment point, the OpenMP compiler has to insert a code fragment that performs the data and work redistribution such that it fits the new number of PEs. We will now cover the code generation templates and discuss the way in which each of the templates and the generated code interact with each other.

To avoid code explosion and to keep the generated code simple, the compiler only emits a so-called reparallelization stub for each adjustment point instead of emitting the complete reparallelization code. It is the stub that checks if a reparallelization has to be performed and triggers the appropriate functions of the OpenMP runtime to actually perform the reparallelization. Another advantage is that this solution yields a clean separation of the code transformation templates in the compiler and the implementation of reparallelization in the runtime system. The reparallelization runtime can be implemented in Java with a high level of abstraction, while the low-level details (e.g. live variable analysis, insertion of copy operations, etc.) that have to be implemented in the compiler back-end can be kept separate from the high-level parts.
4 Reparallelization of OpenMP Constructs

4.7.1 Code Generation Templates for Adjustment Points

During compilation of OpenMP constructs that contain an adjustment point, the OpenMP compiler has to apply different code transformation templates that compile the adjustment point to executable code. The tasks that need to be performed are unique to the considered OpenMP construct. While removing a thread in an OpenMP work-sharing construct produces a partial chunk descriptor, removing a thread from a parallel region only causes termination of the corresponding thread. Similarly, adding a thread to a work-sharing construct implies a redistribution of work, whereas adding threads to parallel regions implies loading of a thread context that provides correct values for live variables.

A closer look at the code templates reveals that a common code template can be factored out. Figure 4.11 shows Java-like pseudo-code that is emitted by the compiler for each adjustment point. Later in this section, we will refine the skeleton to instantiate the code templates for adjustment points in different kinds of OpenMP constructs (parallel and for).

The skeleton consists of three parts. First, the emitted code (lines 1–2) queries an adjustment strategy object. If no change is requested, the rest of the reparallelization stub is skipped and execution continues after the adjustment point. If there is a reparallelization request and the current thread has to be removed, a partial chunk descriptor is created and the current thread is terminated (lines 3–5). Lines 6–10 check if the thread count changes, instruct the master thread to create the new threads (if needed), and adjust the internal data structures of the OpenMP runtime. The purpose of the START_i and END_i label is explained, when we refine the code skeletons.

The reason for using the indirection of an adjustment strategy object is for additional flexibility. It allows to change the behavior of the reparallelization runtime without the need to change and recompile the OpenMP compiler and the reparallelization runtime. The strategy pattern is generally used to decouple the usage of an algorithm and the implementation of it [74]. Hence, the implementation can be changed without affecting the code locations that use the algorithm. With additional OpenMP runtime functions (e.g. setAdjustStrategy in Figure 2.6) it is possible to register implementation-specific or even user-defined strategies that influence the reparallelization of parallel regions. For example, if the default adjustment strategy does not fit
4.7 Compilation of Adjustment Points

```java
strategy = team.getAdjustStrategy();
if (strategy.adjustmentRequested()) {
    if (strategy.removeSelf(threadID)) {
        doTermination();
    }
else if (threadId() == 0 && changedThreadCount()) {
    strategy.createThreads();
    adjustDataStructures();
}
finishAdjustment();
goto END;i
}
start,i:
END;i:
```

Figure 4.11: Generic skeleton of the code templates for adjustment points.

the thread placement of the current machine, the strategy can be replaced by a better one. For instance, the user can provide an adjustment strategy that uses application-specific information to maintain a higher data locality if threads are added or removed. We accept the negligible performance penalty of two indirect function calls to gain more flexibility in tuning the reparallelization runtime in order to better fit the applications’ needs.

When the thread count has to be changed during a reparallelization (we skip the question about the source of this information for now, see Section 5.7.4), the adjustment strategy receives the old number before the reparallelization and the new number of threads after the reparallelization. Using this information, the strategy object can select threads for removal or it can select processing elements for which new threads have to be created. As each thread is responsible for terminating itself if needed, each thread queries the strategy object and asks for its removal. Depending on the thread ID, the strategy object decides for each individual thread whether or not it should be terminated. Adding threads to the OpenMP construct is left to the master thread. If threads have to be added to the execution, the master enters the strategy object. The strategy object then decides on which CPUs on the machine or nodes of a cluster new threads have to be created and spawns the threads on the target PEs.
4 Reparallelization of OpenMP Constructs

**Figure 4.12: Code templates for adjustment points in work-sharing constructs.**

**Work-sharing constructs**

The reparallelization of an OpenMP work-sharing construct implies a redistribution of the work according to the requirements of the scheduling clause at the construct. We have already described the tasks that have to be performed for the different types of loop schedules (see Section 4.4.4). The algorithms that redistribute the loop are triggered by the `adjustDataStructures` statement in the pseudo-code fragment of Figure 4.12.

The generic code skeleton has to be extended as shown in Figure 4.12; the added code is marked with “*”. If a thread is instructed to terminate, it first stores the partial reduction value (if a reduction is performed, lines 4–5). It then creates the partial chunk descriptor for the currently active loop chunk. Adding a thread can be divided into two distinct cases. First, a thread is added to the work-sharing construct and it takes on a partial chunk descriptor to process an already started chunk that is not yet finished (line 17). Using the branch table for the construct, the added thread starts processing at the label `START_\text{i}`, where \text{i} is the unique number for the adjustment point in the construct. Second, if the added thread does not take over a partial
4.7 Compilation of Adjustment Points

```java
1. barrier.block();
2. strategy = team.getAdjustStrategy();
3. if (strategy.adjustmentRequested()) {
4.     if (threadID == 0)
5.         storeLiveVariables();
6.         if (strategy.removeSelf(threadID))
7.             doTermination();
8.     else if (threadID == 0
9.                && changedThreadCount()) {
10.        strategy.createThreads();
11.        adjustDataStructures();
12.    }
13.    barrier.block();
14.    finishAdjustment();
15.    barrier.block();
16. }
17. goto END_i
18. START_i:
19. loadLiveVariablesFromMasterTemplate();
20. END_i:
```

Figure 4.13: Code templates for adjustment points in parallel regions.

chunk, it starts processing at the beginning of the construct after the master thread performed the work redistribution.

**Parallel regions**

Parallel regions need to be handled differently, as there are no loop chunks or partial chunks that need to be assigned to threads. If a thread is removed, its computation stops and no thread takes over the work. If a thread is added, the new thread starts at the adjustment point and executes the code located after the adjustment point. An added thread also needs to receive a valid thread context to initialize live variables with correct values.

These features are implemented by the code template shown in Figure 4.13. Statements added to the generic skeleton of Figure 4.11 are again marked with "*". Before the reparallelization stub of the currently active adjustment point is entered, all threads are collected in a barrier (line 1) to ensure that each thread’s path of execution is at the same code location. This is important for lines 4–5 that create the copy of the master thread’s live
variables. Otherwise, the threads of the team might be at different locations of the program when the adjustment occurs and the computational state might be ambiguous. This causes undefined behavior, since the set of live variables is not necessarily equal at all adjustment points.

If a thread has to be removed, it is terminated and its execution path is discarded. If a new thread is created, it starts execution at the `START_i` label and initializes its live variables (line 19) created earlier by retrieving the values from the master thread’s copy. If an adjustment is requested, the two barriers in line 13 and 15 are entered to ensure that the newly created threads do not start execution before the adjustment is completed and all data structures are up to date. The second barrier (line 13) guarantees that all threads have passed the `adjustmentRequested` and are informed about the reparallelization. The second barrier (line 15) assures that threads only exit the adjustment point after the internal data structures have been updated. Otherwise, threads might perform operations with only partially updated data structures. Thus, if one of the barriers was omitted, various scenarios of race conditions that cause undefined behavior may occur. For example, it could be possible that the data structures are already updated while one thread still is not informed about the reparallelization.

### 4.7.2 Loop Exit Barrier

To ensure that all threads of a team have finished parallel execution before continuing sequential execution, all OpenMP constructs by default involve a barrier synchronization at the end of the parallel execution. With the `nowait` clause, OpenMP offers the programmer a means to avoid the barrier synchronization at the end of a work-sharing construct if it is not a compound `parallel for` construct. If the barrier is omitted, the OpenMP implementation still has to guarantee that the loop is completely executed.

Regular OpenMP implementations do not create new chunks while the parallelized loop runs. For a `static` schedule without explicit chunk size, the loop’s iteration space is divided such that each thread receives exactly one chunk for execution. If each thread exits the construct as soon as it has finished execution of its chunk, all iterations of the loop have been computed. The same argument holds for a static schedule with explicit chunk size, as each thread receives a set of chunks to work on. After it has completed the last chunk, it will not receive new chunks for execution. Although with
4.7 Compilation of Adjustment Points

dynamic or guided schedules the loop chunks are distributed on demand, a thread can exit the loop as soon as it does not receive an unprocessed chunk. As no new chunks can be created during execution, the work pile of unprocessed chunks diminishes as execution proceeds. If a thread sees an empty work pile, the loop has been processed completely and the thread may exit the loop although other threads may still work on a chunk.

With adjustment points added to the loop, partial chunk descriptors may be created to postpone the execution of chunks that have been assigned to terminated threads. As the remaining threads do not necessarily know about the postponed chunks, there is no guarantee that the complete iteration space of the loop has been processed when the last thread exits the loop. Hence, the nowait clause has to be disallowed if the loop contains an adjustment point. A regular barrier synchronization does not provide correct semantics and does not guarantee that all iterations of the loop have been processed. While a barrier is left as soon as all threads have reached the barrier, the barrier has to prevent the threads from exiting the loop before all chunks and all partial chunk descriptors have been processed.

The control flow of the loop exit barrier is depicted in Figure 4.14. The upper part shows the control flow of a parallel loop as it is emitted by the OpenMP compiler for regular loops with a nowait clause. Before it leaves the loop, a thread checks if there are any chunks left that have been assigned to it. If there are no chunks left for execution, the thread exits the loop and enters the code that is located after the work-sharing construct. This corresponds to the while loop shown in Figure 3.9 in Section 3.4.3.

The lower part of Figure 4.14 shows the loop exit barrier and is only used for loops that contain at least one adjustment point. The barrier first tests if the loop was completed by checking (1) if all chunks have been completed and (2) if all threads have reached the barrier. If so, the loop is exited and execution continues with the code after the current loop. If the loop has not been finished yet, the barrier checks if any postponed chunks from terminated threads exist. These chunks are assigned to threads that just tried to exit the loop through the barrier. If no unfinished (partial) chunks exist, the thread blocks and waits for a notification. Blocked threads are notified when a reparallelization occurs in the work-sharing construct and new postponed chunks have been created, if the blocked thread should be terminated, or the other threads have reached the barrier as well and the barrier can be torn down.
4 Reparallelization of OpenMP Constructs

Figure 4.14: Reparallelization barrier at work-sharing constructs.

4.8 Summary

This chapter has described a novel approach to alter the degree of parallelism of an OpenMP program while it is actively executing a parallel region. Changing the degree of parallelism is crucial for migration, as the target environment may provide a different number of PEs than the source system. Non-malleable programs would either overload the system by placing too many SUEs on a too small number of PEs or might not utilize all of the available PEs. If the count changes after a migration, the application has to (automatically) adapt to the new number of available PEs.

It is desirable that the application automatically adapts to the new PE count, that is, the programmer should not be concerned with making the application
malleable. Our solution provides an automatic, compiler-supported means to alter the degree of parallelism of OpenMP applications without the need to change the application’s code. At so-called adjustment points, the degree of parallelism of the currently executed parallel region or work-sharing construct is altered. Adjustment points are introduced by the \texttt{adjust} directive, which we add to the existing set of OpenMP/Java directives. The \texttt{adjust} directive may either be added automatically or manually if desired.

The compiler automatically adds code that performs the reparallelization and adjusts the data/work distribution accordingly. We provide redistribution schemes for each kind of OpenMP construct. Our approach transparently redistributes static loop schedules (with and without user-defined chunk sizes), dynamic and guided loop schedules, and parallel regions. Parallel \texttt{sections} are transformed to a parallel loop and are handled by the reparallelization technique for work-sharing constructs. Section 7.4 will show that the overhead of the added reparallelization stubs is below 5\% on average.

We have identified limitations of this approach and discussed the severity of the programming-model restrictions imposed by these limitations. Reparallelization is only feasible for pure OpenMP programs, that is, OpenMP programs that use OpenMP for both thread creation and data/work distribution. Reparallelization implies a semantic change for parallel regions when the thread count is changed. If the semantics may not be altered, the programmer can prohibit reparallelization by adding the \texttt{adjust(none)} clause at the level of the \texttt{parallel} construct. Finally, we have exposed data locality issues that might occur on systems with NUMA or COMA characteristics.
5 Heterogeneous Distributed Checkpointing

Our migration package relies on checkpointing as a means to transport the application’s state from the source system to the target system. With checkpointing, an application saves its computational state on a persistent storage device (e.g. disks, network shares on a server). Later, the saved state can be retrieved from the storage device and the application continues from the point in the code at which the checkpoint was created.

In many cases, checkpointing techniques are used to add fault-tolerance to applications. This is especially needed for applications that have longer expected runtimes than the Mean Time Between Failure (MTBF) of the execution environment. For example, the MTBF of the BlueGene/L system with 65,536 nodes is about ten days [2]. For applications that run longer than ten days, one can expect a failure in the execution environment before the application successfully completes its computation. If a failure happens, the application has to be restarted, as the computational state is lost when the application is prematurely terminated. With checkpointing, however, the application can save its computational state to disk in regular intervals. If the execution environment is affected by a failure and the application cannot continue computation and thus has to be aborted, the application may resume from the latest checkpoint and continue execution from where the checkpoint was created. Because the application preserves already computed (intermediate) results in the checkpoint, there is no need to re-compute from scratch and less computing power is lost in case of failures.

While the approach shown in this chapter could also be used to implement checkpointing for fault-tolerance, we restrict ourselves to checkpointing to enable application migration between different and heterogeneous systems of
5.1 State of the Art

Using checkpointing techniques to make an application resilient to faults in the computing environment is not new. There are various different approaches with different characteristics and different aims. Surveys on checkpointing techniques including their classification can be found in \[110, 165\].

Table 5.1: Classification of checkpointing techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Negligible overhead</th>
<th>Consistent checkpoint creation</th>
<th>Redundancy free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Incremental</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Independent</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Coordinated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Legend:
✓ full match
✗ no match

a computational Grid. Ignoring fault-tolerance, we can optimize our checkpointing system to optimally support migration.

The chapter first presents the state of the art in (distributed) checkpointing (Section 5.1) and covers the design decisions that govern our checkpointing approach (Section 5.2). We then give a short introduction into the problems a heterogeneous checkpointing solution has to solve (Section 5.3). Section 5.4 describes our generic checkpoint format. In Section 5.5, we present how a multi-threaded single-process application can be checkpointed. Section 5.6 presents a distributed checkpointing algorithm that checkpoints loosely coupled multi-process applications. After the presentation of an algorithm to store Jackal’s DSM space (Section 5.7) to disk, the chapter closes with a discussion of the impacts of checkpointing to application semantics in Section 5.8.
Table 5.2: Classification of checkpointing packages.

<table>
<thead>
<tr>
<th>System</th>
<th>No OS modification</th>
<th>Platform-independent</th>
<th>Transparent</th>
<th>Distributed</th>
<th>Negligible overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>libchkpt</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>BLCR</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dome</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Porch</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>[28]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JESSICA2</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>PREACHES</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Cactus</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>DGET</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>P-GRADE</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>GrADS</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[5]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[145]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LAM/MPI</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CoCheck</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[45]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MOSIX</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sprite</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMPI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[69]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jackal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Legend:
✓ supported
× unsupported
We restrict ourselves to a discussion of selected checkpointing and migration solutions that help to understand the design decisions made in our checkpointing system. Table 5.1 lists four types of checkpointing techniques for multi-process applications and assesses their applicability. Table 5.2 classifies a selection of projects with respect to the requirements for our migration solution (see Section 5.2).

In Section 5.1.1, we discuss the checkpointing packages listed in Table 5.2. Section 5.1.2 presents other solutions for application migration.

### 5.1.1 Checkpointing Solutions

Checkpointing techniques that target fault-tolerance, such as incremental checkpointing [4], asynchronous checkpointing [31], or independent checkpointing [65], generally strive to minimize the impact of checkpointing on application performance during normal execution.

Incremental checkpointing stores only the differential state, compared to the preceding checkpoint. Since a checkpoint only comprises a differential state, all checkpoints must be transferred to the target system during a migration. However, the total size of the incremental checkpoints is larger than the size of a single checkpoint taken at migration time. If a data item is modified between two incremental checkpoint creations, the prior checkpoint redundantly stores data that is no longer valid. Our checkpointing package keeps the checkpoint as small as possible by checkpointing the application’s current computational state only.

Asynchronous checkpointing approaches write changed data to disk on demand while the application continues to execute and, thus, cannot guarantee that a checkpoint contains the complete application state at a given point in time. Moreover, the application’s performance is reduced during normal operation, as the checkpointing runtime concurrently writes the checkpoint to disk and consumes computing power. Our solution only creates a single checkpoint for a migration and thus keeps the impact of checkpointing as low as possible.

The loosely coupled nature of multi-process applications that consist of several interacting processes, requires special algorithms to create a (distributed) checkpoint. While the checkpointing system can checkpoint each process individually, it must ensure that the local checkpoints together form
a consistent global checkpoint. The theoretical background of multi-process (distributed) algorithms is discussed in Section 5.6.1. Several projects handle multi-process checkpoints in different ways (see below).

Independent approaches do not necessarily guarantee that each checkpoint is consistent and so a consistent state has to be found at restart time. Section 5.6.1 discusses the details of independent checkpointing algorithms and presents reasons why these algorithms cannot be used for migration. Only a coordinated checkpointing algorithm is sufficient for application migration.

Although our system targets distributed multi-process applications, it is instructive to investigate techniques for single-process checkpointing. All multi-process checkpointing packages rely on such techniques to save the individual processes of a distributed application. The state of the application that is executed in the process needs to be stored in a checkpoint such that the application can resume operation later on.

Checkpointing solutions such as libchkpt [160] and BLCR [60] checkpoint single process applications. Libchkpt is a non-transparent checkpointing solution implemented as a library that is linked into the application. Libchkpt creates incremental checkpoints, that is, each checkpoint only comprises the difference to the last checkpoint. BLCR extends the Linux kernel with a checkpointing module that writes the process image to the checkpoint upon request; the application does not necessarily have to be checkpointing-aware. Similar to our solution, BLCR stops all threads in the process to avoid race conditions during checkpointing. Both libchkpt and BLCR only create platform-dependent checkpoints, as they copy the process image to the checkpoint file. Kernel modifications have to be avoided, since clusters typically use special versions of OS kernels with support for specialized hardware, such as high-speed interconnects.

Platform-independent checkpointing solutions save the computational state of an application on one architecture and resume the application on a different architecture. As the computational state of an application is highly machine-dependent, a platform-independent solution must convert the state from its physical representation into a portable format. There are several approaches, which we discuss next.

Dome [17] and Porch [162] support platform-independent checkpointing. Both employ a preprocessor that transforms the application code and augments each function of the program with additional calls to the checkpoint-
5.1 State of the Art

An extended JVM [28] and JESSICA2 [213] support platform-independent checkpointing of Java threads. Both rely on augmented Just-in-Time (JIT) compilers to analyze the optimized code and to deduce the layout of the byte-code stack from the layout of the physical stack. The major problem of this approach is that it has to respect all (JIT) compiler optimizations potentially applied to the code. Since compiler transformations are not necessarily reversible, a generic solution is hard to achieve. However, as the byte-code stack of Java is platform-independent, so are the checkpoints. While both solutions are limited to checkpointing a single thread, our checkpointing solution migrates multiple processes. Furthermore, our solution is less invasive as only high-level compiler passes have to be modified, whereas low-level passes may be left unmodified [204].

None of the above projects supports checkpointing of multi-process applications that consist of several interacting processes. The loosely coupled nature of multi-process applications requires special algorithms to create a (distributed) checkpoint from the local checkpoints of the individual processes. While the checkpointing system can checkpoint each process separately, it must ensure that the local checkpoints together constitute a consistent global checkpoint. A short introduction into multi-process (distributed) checkpointing algorithms is presented in Section 5.6.1. As there are several different ways to handle multi-process checkpoints, we now investigate a selection of multi-process checkpointing packages.

PREACHES [180] is a checkpointing solution that sends the computational state of a master process to slave processes for checkpointing. To achieve platform-independent checkpoints, a slave must be running on all supported architectures. In addition, PREACHES relies on synchronous communication and ensures checkpoint consistency by means of a global barrier. Our checkpointing solution reuses the concept of a global barrier, but provides consistent checkpoints in presence of asynchronous communication. Additionally, checkpoints can be created without the necessity of running a checkpointing slave on all supported architectures.
5 Heterogeneous Distributed Checkpointing

Cactus [76], DGET [63], P-GRADE [121], and GrADS [200] support checkpointing of multi-process (distributed) applications. The application has to extend a framework by using callbacks from the framework into application code [12, 63] or functions must be called to register data that constitutes the computational state of the application. Both solutions do not fit our requirements, as they are not transparent to programmers and add high overheads to the application. For migration, we desire to keep the overhead of checkpointing as low as possible and do not accept invasive changes to the application’s code base. Our compiler creates heterogeneous checkpoints without the need to register the checkpoint data by means of API functions.

Multi-process (distributed) checkpoints are also discussed in [5, 145, 170, 181]. The compiler of [5] deduces locations of checkpoint initiation by examining the application’s control flow. Checkpoint creations are placed at program locations that are expected to yield a consistent global state. The checkpointing system in [145] employs synchronized clocks to maintain a consistent global state for a checkpoint. LAM/MPI [170] offers multi-process checkpointing using BLCR as checkpointing implementation and a coordinated (stop-the-world) algorithm (see Section 5.6.1); CoCheck [181] also uses a coordinated checkpointing algorithm. Similar to LAM/MPI and CoCheck, our checkpointing solution relies on a coordinated algorithm.

5.1.2 Migration Solutions

Live migration [45] transfers an OS instance from one host machine to another one without shutting down the OS instance. The migration system moves the contents of the main memory and the state of local resources to the target system without stopping the execution of the OS and the active processes. Live migration seems to provide an effective means for application migration. However, the discussion of Section 5.2.2 reveals that live migration is not applicable for application migration in Grid environments.

Migration of Java threads is implemented in [28] and in JESSICA2. A Java thread is moved between instances of JVMs by checkpointing the thread’s state and shipping the state to the remote instance of the target JVM. In contrast to our work, migration of a complete application is not supported.
MOSIX [22] and Sprite [57] migrate (multi-threaded) processes from one machine to another. During migration, a process stub is left behind at the old node, accessing immobile resources such as open files and network connections. In contrast to our solution, both systems cannot migrate between machines of different types. MOSIX and Sprite can only migrate standard, non-cluster OpenMP applications for shared memory hardware. Our work extends the process migration facility to provide migration of multi-process, multi-threaded applications that are executed on a DSM system.

There is previous work that focuses on the migration of applications from one cluster of the Grid to another. GrADS, P-GRADE, and AMPI [95] use checkpointing techniques and migrate applications between clusters. GrADS enforces its own programming model [25]. P-GRADE migrates MPI/PVM codes and AMPI is limited to its MPI implementation. The same holds for the mobile MPI programs of [69]. At the time of this writing, migration packages suited for OpenMP applications (on DSM systems) are not existing. Additionally, none of the above projects has been applied to OpenMP applications. Thus, we provide a genuine solution for shipping OpenMP applications between clusters of a computational Grid.

Our checkpointing package targets application migration and is the only available solution that does not need kernel modifications to create checkpoints. The checkpoints are stored platform-independently and at the same time the high overhead of other platform-independent checkpointing approaches is avoided. A coordinated stop-the-world algorithm ensures that the checkpoint is always consistent and as small as possible. As our solution avoids expensive calls to API functions during normal operation of the application and thus its overhead is almost negligible.

5.2 Checkpoint-based Migration

A checkpointing solution for migrating applications between clusters needs to fulfill other requirements than a solution for providing fault-tolerance to applications. Let us now focus on the requirements for both types of checkpointing and discuss the differences between these two aims. We will also compare our solution to approaches that migrate applications by means of live migration. With live migration, the application continues to run while it is moved to the target system in a piecewise manner.
5.2.1 Requirements

Checkpointing for fault-tolerance is governed by the goal to make applications resilient to failures. Any failure in the computing environment shall be compensated by periodically saving checkpoint information to a persistent storage device. It is desirable that, in case of a failure, as much computational work as possible is preserved. Hence, creation of checkpoints accompanies execution as the application executes and potentially creates various checkpoints during its lifetime.

The more checkpoints the application creates, the less computation needs to be re-performed in case the application restarts from a checkpoint. As taking a checkpoint is a costly operation, a tradeoff has to be found that minimizes the costs for checkpointing and that maximizes resilience of the application against faults. Generally, it is desired that the application proceeds execution until the computation is finished and the stored checkpoints are never actually used.

In contrast, if an application checkpoints and subsequently migrates to the target system, it only creates a single checkpoint and temporarily stores it on disk. The application always terminates after the checkpoint has been created, the state is transferred to the target system, and the application resumes from the transferred checkpoint data on the target system. Only complete checkpoints provide an effective means to transport the state of an application. A checkpoint needs to be consistent at any time, since otherwise the application is not able to resume operation on the target system.

In addition to consistency, we require our checkpoint solution to be transparent, that is, the programmer must not be concerned with supporting checkpointing in the application by placing calls to, for example, runtime functions. The compiler should identify code locations at which state saving may occur. If no checkpoint is created, the performance penalty imposed by the checkpointing runtime should be as small as possible.

Writing the computational state to disk should be as fast as possible, as parallel applications typically exhibit a large memory footprint. At the same time, the checkpoint data must be as small as possible, since the data has to be transferred to the target system over low-bandwidth Wide Area Network (WAN) connections.
5.2 Checkpoint-based Migration

Only user-land checkpointing solutions that do not need kernel modifications suit the needs for application migration. First, it is virtually impossible to provide a checkpointing driver for all past and recent versions of the OS kernels that drive a cluster’s nodes. Second, parts of the Grid are typically controlled by different parties and no global steering exists. Hence, it is impossible to require a certain software module (e.g. the checkpoint-enabled kernel) on all accessible machines. Third, most clusters use specialized hardware that needs OS device drivers for a certain OS version to function properly. For these reasons, the kernel cannot be modified as required.

Finally, the heterogeneous structure of a computational Grid must be respected. The checkpoint data has to be saved in a format that is independent of the architecture a checkpoint was taken on. Otherwise, it would be impossible to resume the application on another architecture. We cover this issue in more detail in Section 5.3.

5.2.2 Checkpointing versus Live Migration

Applying checkpointing to solve the problem of migrating an application from one system to another, implies that the application is completely inactive while it is being migrated. Otherwise a consistent checkpoint cannot be guaranteed if rollback actions have to be avoided at restart. The application’s execution is frozen by saving its computational state in a checkpoint. The checkpoint is then copied to the target system and the application is resumed from the stored state information.

Another possible solution to migration is to migrate only one process instance at a time [45]. Live migration is used to transfer an OS instance from one host machine to another one without shutting down the OS instance. Using the same techniques, applications can be continuously migrated to the target system without stopping computation. At the target system, a new process is launched that receives the data and the threads of an existing process of the application. After the data and threads have been moved to the newly created process, the old process is empty and can be removed. However, network connections may still be active and have to be re-routed. Adding and removing one process at a time continues until all processes of the application have been migrated to the target system. Live migration is not an option for computation migration in computational Grids, because of two reasons.
First, most parallel applications are not pleasingly parallel and thus communicate frequently between the processes that constitute the application. Since live migration migrates processes while the application is active, the application’s performance deteriorates. Messages between the processes on the source system of the migration and the target system have to cross the high-latency, low-bandwidth WAN interconnect. Hence, stopping the application causes roughly the same overall performance penalty as a live migration approach. A stop-the-world approach is much easier to implement, as no race conditions occur while the migration is conducted.

Second, and most importantly, continuous migration approaches do not blend well with the current way of reserving computing time at a cluster. As during the live migration processes have to exist on the source system as well as on the target system, the old and the new (possibly remote) reservation for the application have to be active at the same time. In a Grid environment, no central scheduling instance in general exists that provides co-scheduling of the source and target reservation. Meta-schedulers that guarantee co-scheduling are still an open research topic [53, 104, 129]. Freezing the application state in a checkpoint thus is the only feasible solution to the inter-cluster migration problem.

5.3 Heterogeneous Checkpointing

The computing sites accessible in a computational Grid can be of a different architecture. Each of the systems does not only potentially differ in the type of the nodes’ CPUs (e.g. IA-32, PowerPC, EM64T), but each of the computing sites may also run a different operating system or a different version of the same operating system. For example, a cluster at one computing center may consist of nodes with IA-32 CPUs running Linux kernel v2.6.22, while another cluster is equipped with EM64T CPUs and uses Linux kernel v2.6.5. As we will see, the version of the OS kernel as well as the machine’s architecture influence the application state.

Before we can give an impression of why transferring the application state between systems is a non-trivial problem, we have to investigate what constitutes the application state. From an application point of view, the following contributes to the application’s state.
5.3 Heterogeneous Checkpointing

Parts of the computational state of the currently active method are described by the values of local variables defined in the method’s scope. In addition, the arguments passed to a method also contain values that might prove important for computation. If the method accesses static fields of a class, these static fields also store a subset of the computational state. Finally, the objects that reside in the application’s heap account for the largest fraction of the computational state.

From a system programming point of view, the operating system has to maintain data structures that describe resources the application is currently using [173]. The OS keeps a per-process PCB that stores information about the application’s process. The PCB contains information about the virtual memory table of the process, open files, open network connections, and many more. The format of these data structures is highly dependent on the version of the operating system and the machine’s architecture. Hence, the internal data structures of the OS cannot be directly saved to achieve a platform-independent checkpoint. During restoration, a different operating system is likely to maintain different data structures.

From a compiler point of view [7, 16], the application state is distributed over a set of machine-dependent parts of the memory hierarchy. A compiler assigns live variables that are currently needed for the computation to registers as often as possible. A live variable that is currently not assigned to a register resides on the method’s stack frame on the stack. Objects are allocated on the application’s heap; each field of a particular object is assigned an offset relative to the object’s address. The call stack and the heap reside in the so-called data segment [106]. For static fields, the compiler usually allocates memory in the BSS segment, which is automatically initialized with zero values. If the static field has to be initialized, the compiler generates a static initializer that assigns the non-zero value to the static field.

Figure 5.1 shows an example code with a method caller that takes two parameters a and b. A third parameter is automatically added by the compiler to provide for the implicit this reference [16]. The method defines a variable i that is used as the counter in the for loop. The Application Binary Interface (ABI) of the architecture prescribes how a compiler maps variables to physical storage locations.

Figure 5.2(a) depicts an IA-32 allocation scheme [195] to map the variables of caller to physical storage locations. The arguments for caller are expected to be located on the stack. Traditionally, arguments are pushed in reverse
class RegisterAllocationExample {
    void caller(int a, long b) {
        for (int i = 0; i < a; i++) {
            callee(b);
        }
    }
}

void callee(long x) {
    // some computation
}

Figure 5.1: Example method with three variables.

(a) Allocation on IA-32.  (b) Allocation on EM64T.

Figure 5.2: Allocation of variables in the example of Figure 5.1 on IA-32 and EM64T.
5.3 Heterogeneous Checkpointing

order, that is, the last argument is pushed on the stack first. Argument \( b \) is split into two 32 bit words (high and low), since \( b \) is a 64 bit variable. The saved return address (\%eip) and the saved base pointer (\%ebp) immediately follow the arguments. The loop counter is assigned to the \%esi register.

On EM64T architectures, the physical storage of Figure 5.2(b) is used. The ABI for EM64T [137] requires that arguments have to be passed using registers. Hence, the Jackal compiler uses the registers \%rbx, \%rsi, and \%rdi to pass the values of \( a, b, \) and \texttt{this}, respectively. As \( a \) is a 32 bit variable, its value is passed using the lower half of register \%rbx (\%ebx). The only data stored on the stack is the return address (\%rip) and the saved base pointer (\%rbp). The loop counter is assigned to register \%r8.

As this example shows, it is not possible to create a 1:1 copy of the application’s memory image on disk and to resume the created image on another machine of a different architecture. Such 1:1 copies would need an exactly matching physical storage layout on all architectures, which is obviously impossible to achieve. Checkpointing solutions such BLCR [60] or libchkpt [160] create such images by writing the application’s memory image to disk. However, if the operating system version on the target machine or the architecture changes, the checkpoint cannot be resumed on the target machine. Converting the checkpoint image is impossible, as the memory image does not contain any type information that is necessary to analyze the data stored in the checkpoint.

To create a heterogeneous checkpoint, three different solutions arise. First, an application can register its data along with the corresponding type information. When a checkpoint is created, the checkpointing algorithm traverses the set of registered data, analyzes the attached type information, converts the data to a generic format, and writes the converted data to disk. Porch [162] is an example of such a checkpointing solution. The Porch compiler automatically augments the application with calls that register the data of the application and write checkpoints.

Second, a checkpointing algorithm can try to analyze the physical storage layout of the application and deduce a generic stack-frame layout from it. However, optimization techniques applied to the application’s code make it difficult to deduce the memory layout. The checkpointing algorithm thus needs to de-optimize the application code, analyze the physical data layout [44], and create the platform-independent checkpoint from it. An example are the thread migration and checkpointing techniques of [28, 213].
Recent JIT compilers replace active methods on the stack by similar techniques (called on-stack replacement [176]).

The third solution is a trade-off between the first and the second solution. It is based on the observation that only when a checkpoint is created, the type information is needed and the data has to be converted to a platform-independent format. As Java is a type-safe language, the complete type information of each variable and object is available to the compiler at compile time. Hence, the compiler can provide the type information to the checkpointing algorithm statically, that is, without the need to register the types at runtime. The checkpointing algorithm uses the static type information and converts the data along with its type information into a machine-independent checkpoint image.

Our approach is based on the third solution. The compiler determines the types for all variables and objects and provides for functions to map the platform-dependent physical storage locations to a generic checkpoint format that is used for transferring the application state. Our generic format relies on a unique name that is assigned to a variable and that is associated with the variable’s value during checkpointing. The next section discusses Jackal’s compiler extension responsible for creating the unique identifiers.

5.4 Unique Descriptor Strings

To create the generic checkpoint format, a mechanism is required that in principle re-associates the physical addresses of variables (i.e., registers, stack-frame addresses) with their corresponding source-code level variable name. Unfortunately, optimization passes of a compiler introduce new variables (without a name in the source code) or remove variables; with increasing levels of optimization, source-code naming information is lost. Therefore, instead of using source-level names, our compiler generates so-called Unique Descriptor Strings (UDS) that uniquely describe a variable. A UDS is kept fixed when a variable is mapped to a physical storage location and thus provides a platform-independent name for any variable that is part of the computational state of the application.
5.4.1 Construction of Unique Descriptor Strings

Creating the UDS for a variable is based on the following assumptions [204]. Suppose a method \( M \) calls another method \( N \) at a particular call instruction \( C \). Then a compiler can identify the set of variables \( V = \{v_1, v_2, \ldots, v_n\} \) that are live at the call site \( C \). It is the set of live variables in \( M \) at \( C \) that determines the result of the computation when \( M \) proceeds with calling \( N \). Hence, saving the variables in the checkpoint conserves the computational state of \( M \) at call site \( C \).

To restore the computational state on a different architecture, the set of live variables \( V \) at \( C \) has to be the same on all architectures. This can be ensured by using the same compiler options for compiling the application code, effectively causing the same transformation of the application code for all architectures. Keeping the set of live variables fixed for a call site \( C \), the compiler can emit a mapping function (the checkpointer) that maps the platform-specific stack-frame layout and register set to a generic stack-frame format when a checkpoint is created. The inverse function (the uncheckpointer) that restores a native stack frame from the generic format can also be created by the compiler.

The focus will now be set on how the Jackal compiler creates the UDS for a particular variable \( v_i \). For a live variable \( v_i \) at the call site \( C \), there is a given preceding computation \( H \) that fully determines the value of \( v_i \). Since the same source code is compiled, the computation \( H \) has to be the same on all architectures. Otherwise, the application would have different semantics on different architectures. \( H \) uses the same operations on all architectures. Low-level transformations (e.g. peep-hole optimizations [7]), however, may change these operations to more efficient ones. For example, a multiplication operation may be replaced by a shift operation or several addition operations used for address calculation can be changed to a single “load effective address” instruction. While the sequence of operations is not the same after the peep-hole optimization has been applied, the semantics of the computation remain unchanged.

Considering the above observation for the computation of a variable \( v_i \)’s value, the computation affecting \( v_i \) can be conceptually separated from the rest of the instruction stream of the surrounding method by creating a new function \( H' \) that only contains the operations affecting \( v_i \). Hence, \( H' \) uniquely describes the computation leading to the value of \( v_i \) at \( C \) in
a platform-independent way. The checkpointer function created for the call site \( C \) can store \( H' \) along with a platform-independent representation of \( v_i \)’s current value. Likewise, the uncheckpointer can read \( H' \) from disk along with the value of \( v_i \) and reconstruct \( v_i \) at its machine-dependent physical storage location.

### 5.4.2 UDS Construction Rules

In [204], the above rationale and the following algorithm for creating the UDS of an variable were presented. The algorithm uses the sequence of operations \( H' \) as input and constructs the UDS as a string representation of \( H' \). For unary or binary operations, the algorithm appends the operation performed as a string (e.g. “+” for addition, “-” for subtraction, etc.) followed by the result of a traversal in prefix order of the operation’s operands. If a constant is processed, it is appended to the string by converting the constant’s value into a decimal string. Calls to methods are treated as sort of user-defined operations and the name of the called method is appended as the name of the operation.

The rules that were proposed in [204] are as follows:

1. If the current operation is of the form “\( v = c \)”, where \( v \) is a variable and \( c \) is a constant value (e.g. 0, 0.0, null), then the resulting UDS for variable \( v \) is “C:c”.
2. If the operation is “\( v = w \)” (\( v \) and \( w \) are variables), then assign the UDS of \( w \) to the UDS of \( v \).
3. If the operation is “\( v = \text{call method} \)” generate the UDS “R:n” for \( v \), where \( n \) is the index of the call in all call sites in the containing function.
4. If “\( v = \text{param}(x) \)” is seen, emit “P:n”, where \( n \) is the index of “\( x \)” in the parameter list of the call to the method.
5. If the operation involves a field access “\( e.f \)”, where \( e \) is an expression that evaluates to an object reference and \( f \) is the name of a field of the referenced object, emit “M:f” and recursively continue with the variables in \( e \), and append the resulting UDS.
5.4 Unique Descriptor Strings

6. For any operation “\( v = \text{op}(o_1, o_2, o_3) \)”, create a UDS for \( v \) with contents “\( \text{op} \)”, recursively process \( o_1, o_2, o_3 \), and append the resulting strings to “\( \text{op} \)”. For unary or binary operations, \( o_2 \) and \( o_3 \) may be left blank, respectively.

7. Whenever one of the rules 1 to 6 is triggered and a UDS is created, append “\( @B:n \)”, where \( n \) is the number of the basic block [16] in which the corresponding operation is located.

The computation \( H \) that determines the value of a variable \( v \) and the usage of \( v \) may be spread across the containing method. While this computation can be conceptually described by a method \( H' \) (see above), it would be too expensive to actually splice out \( H \) into \( H' \). However, the compiler can analyze the instruction stream of the containing method and create the UDS from the complete instruction stream.

The last operation of \( H \) that finally computes the value of \( v \) and assigns the new value to it, can be found by standard compiler analysis by searching for the reaching definition of the variable \( v \). Simplified, a definition \( D \) of variable \( v \) is said to reach a usage \( U \) of \( v \), if and only if \( v \) is defined in \( D \) by assigning a value to \( v \) and if there is no other (weak) definition that may overwrite the value of \( v \) between \( D \) and the usage at \( U \) [16].

As a variable may have a set of reaching definitions, the UDS construction algorithm has to provide a means to handle multiple reaching definitions. Our solution is to concatenate the UDS that stems from each individual reaching definition to form a single compound UDS for a particular variable.

Thus, for each variable \( v \) that is live at a call site \( C \) three steps are necessary to compute the UDS for \( v \): (1) The compiler searches for all reaching definitions of \( v \). (2) For each reaching definition, the compilers creates the UDS by analyzing the expression that is used to define \( v \). To do so, the compiler applies the aforementioned rules to the definition’s right hand side. If there are multiple definitions, append each resulting UDS to form the compound UDS. (3) If the definition expression (and therefore the UDS) references other variables, the compiler recursively proceeds with (1) until the resulting UDS does not contain any referenced variables. Obviously, cyclic definitions of variables may occur (see Figure 5.3, line 12). The compiler detects such cycles and terminates the search for definitions by means of a marker entry in the UDS (see below).
Figure 5.3: Example code with live variables to illustrate the creation of UDS.

Figure 5.3 shows an example code to illustrate the application of this algorithm for creating the UDS for a particular variable. At the call site of `createCheckpoint` (line 13) the variables `a` and `b` are live. The method `callee` contains four basic blocks that are numbered from B0 to B3. The call to `createCheckpoint` is automatically generated and added to the code by the Jackal compiler [204].

For the creation of the UDS for variable `b`, the compiler searches for reaching definitions of `b` in line 13. In B0, `b` is defined with the value 1000 (line 10). Hence, the compiler creates the UDS “C:1000” (rule 1) to reflect this assignment of a constant. Rule 7 causes the compiler to append “@B0”, indicating that the assignment is performed in basic block B0. The resulting UDS is “(C:1000@B0)”. As the UDS does not contain any unresolved variable names, the compiler stops and emits the UDS.

The creation of the UDS for `a` is more complex. The compiler searches for the definition of `a` that reaches the call site of `createCheckpoint`. The only reaching definition is the addition and assignment in basic block B1 (line 12). Hence, the compiler creates the preliminary UDS “(+ a (C:1@B1))” (rule 6), after resolving the constant to “(C:1@B1)”. As the UDS contains an unre-
solved reference to variable $a$, the compiler recursively searches for the UDS of the unresolved $a$.

The $a$ used in line 12 has two reaching definitions. First, the value 0 is assigned in line 9. The UDS for this definition is “(C:0@B0)” (rules 1 and 7). Second, $a$’s new value is defined as $a + 1$ in line 12. Applying rules 6 and 7, the resulting UDS is “(+ a (C:1@B1))”. As can be seen, the UDS again contains an unresolved $a$, which leads to an endless recursion. The recursion can be avoided by using an abort marker “Z” in the UDS, if the definition of a variable references the defined variable. If the compiler observes in the recursion that it has to resolve a variable that it already tried to resolve in a higher level of the recursion, it aborts and emits “Z” instead of searching for the UDS. Hence, the UDS for the second reaching definition of $a$ is “(+ (Z@B1) (C:1@1))”. Concatenating both UDS yields “(C:0@B0) (+ (Z@B1) (C:1@1))” for the reaching definitions of $a$ in line 9.

The compiler can now complete the UDS “(+ a (C:1@B1))” and replace the $a$ contained therein with the created UDS. Hence, the final UDS that reflects the complete computation affecting $a$ is “(+ (C:0@B0) (+ (Z@B1) (C:1@1)) (C:1@B1))”.

As the UDS is created statically when the application is compiled, the UDS do not cause any performance penalty during execution of the application. However, the size of the application’s executable is increased, as the compiler emits a table that stores the created UDS together with metadata needed for checkpointing the application (see the following section).

### 5.5 Checkpointing a Single Process

Single processes on traditional operating systems consist of a single process space that is divided into several segments [106]. The text segment contains the binary code of the application and read-only data. The checkpointing algorithm does not need to write this segment to disk, as it is supposed to exist in the executable that is either shipped to the target system or is already installed there. Writable data of the application resides in the data segment, which stores the stacks of the threads, the application heap, and the static fields of classes. In a multi-process DSM application, the checkpoint must also store the state of the DSM protocol (see Section 5.7).
We will now investigate how the stack of a single thread is checkpointed (Section 5.5.1). We extend this technique to multi-thread checkpointing in Section 5.5.2. Section 5.5.3 describes our approach to checkpoint the application’s heap and static fields.

5.5.1 Single-thread Checkpointing

If a thread encounters a call to `createCheckpoint` (e.g. see Figure 5.3, line 13), the computational state of the method that calls `createCheckpoint` has to be saved. For now, we restrict the computational state to the values of live variables and ignore the state of the objects on the application heap. We will come back to this issue in Section 5.5.3.

For the example of Figure 5.3, the computational state at `createCheckpoint` in line 13 consists of the values of the live variables $a$ and $b$, for which the compiler has created the UDS together with a pair of checkpointer and uncheckpointer to store the activation record of callee. Figure 5.4 shows the pair of checkpointers/uncheckpointer that the compiler emits for the example of Figure 5.3. Each of the checkpointers and uncheckpointer utilizes primitive-type checkpointers and uncheckpointer to store and retrieve the values of variables of primitive types.

The checkpointer of caller (on IA-32) in Figure 5.4(a) reads the values of the register `%edi` that stores the value of $a$ and writes the UDS together with $a$’s value to the checkpoint (line 7). It then retrieves and stores the value of variable $b$ (line 9). The UDS that is created for the variables is passed to the primitive-type checkpointer as an argument. In Figure 5.4(b), the uncheckpointer for callee (on EM64T) reads the values of $a$ from the checkpoint and writes its value to the assigned register (line 7). Variable $b$ is restored the same way in line 9.

As callee itself may have been called by another method (caller), the state of caller has to be stored as well. Hence, for each invocation of a method that is root of a call graph and potentially leads to an invocation of `createCheckpoint`, the compiler has to generate a pair of checkpointer and uncheckpointer. Generally, the compiler has to create the checkpointers/uncheckpointer for any method whose stack frame might appear on the thread’s stack. For these methods, the checkpointer/uncheck-
5.5 Checkpointing a Single Process

(a) Checkpointer on IA-32.

```c
void __checkpointer_UdsCreationExample_caller () {
    __ia32_store_int32_mem("(C:2@B0)", %esp, -4);
    __ia32_store_int32_mem("(C:21@B0)", %esp, -8);
}

void __checkpointer_UdsCreationExample_callee () {
    __ia32_store_int32_edi("+(C:0@B0)" \\
                            "(Z@B1) (C:1@1) (C:1@B1))");
    __ia32_store_int32_ebx("(C:1000@B0)");
}
```

(b) Checkpointer on IA-32.

```c
void __checkpointer_UdsCreationExample_caller () {
    __em64t_restore_int32_mem("(C:2@B0)", %esp, -8);
    __em64t_restore_int32_mem("(C:21@B0)", %esp, -16);
}

void __checkpointer_UdsCreationExample_callee () {
    %ebx = __em64t_restore_int32_reg("+(C:0@B0)" \\
                                    "(Z@B1) (C:1@1) (C:1@B1))");
    %r12d = __em64t_restore_int32_reg("(C:1000@B0)");
}
```

(b) Uncheckpointer on EM64T.

Figure 5.4: Checkpointer and an uncheckpointer for the example code in Figure 5.3.
void unwindAndCheckpoint(Stack st) {
    StackFrame sf = st.top();
    while (!sf.isBottomOfStack()) {
        Address pc = sf.getReturnAddress();
        Checkpointer cp = checkpointerTable.find(pc);
        writeMetaInformation(cp.getMethodName(),
                             cp.getCallSiteIndex());
        cp.writeStackFrame(sf.getAddress());
        sf = sf.getParentFrame();
    }
}

Figure 5.5: Pseudo-code for unwinding and checkpointing a single call stack.

void restore(Stack st) {
    ListOfStackFrames l = readCheckpointData();
    CheckpointData cd = l.getLastElement();
    while (cd != null) {
        String method = cd.getMethodName();
        int callSiteIndex = cd.getCallSiteIndex();
        Uncheckpointer ucp =
            uncheckpointerTable.find(method,
                                      callSiteIndex);
        ucp.restoreStackFrame(st.getStackPointer());
        cd = cd.getPrevious();
    }
}

Figure 5.6: Pseudo-code for restoring a single call stack.

pointer pair is emitted for all call sites that might lead to an invocation of createCheckpoint.

When emitting the application’s binary code, the Jackal compiler also emits the checkpointer and uncheckpointer that have been generated during compilation. In addition, the compiler emits a mapping table, that maps the addresses of the call sites to the corresponding checkpointer. A similar table is created for the uncheckpointer. These tables are used at runtime to find the correct checkpointer/uncheckpointer for a given call site of a method whose stack frame is on the stack.
Using the created checkpointers and the mapping table, the runtime system performs stack unwinding to traverse the call stack of the thread, starting at the stack frame that contains the call to createCheckpoint (see Figure 5.5). The return address of the current stack frame is used as an index into the checkerpointer table and to retrieve the corresponding checkpoint to save the stack frame to disk. Then the algorithm proceeds to the next stack frame. The unwinding stops when the last stack frame has been processed.

Restoring the call stack is performed by reading the stack frames from disk (see Figure 5.6). The processing order has to be reversed, as the stack frame that was checkpointed last, has to be restored first. Hence, the runtime system starts restoration with the stack frame checkpointed last. It retrieves the corresponding uncheckerpointer from the uncheckerpointer table. The uncheckerpointer allocates the stack frame, reads the UDS and the values associated with them, and copies the values to the correct stack locations. This process is repeated until the list of to-be-restored stack frames is empty and the complete call stack of the thread has been restored.

This technique keeps the overhead of our checkpointing package as low as possible. During its normal operation, a thread only encounters invocations of createCheckpoint. As most of the time the checkpointing algorithm is inactive, createCheckpoint immediately returns. Thus, the overhead is almost insignificant. Only if a checkpoint is requested, the checkpointing runtime executes the checkpointers and uncheckpointers. Hence, the application only suffers from a performance penalty if a checkpoint is actually created and stored to disk.

### 5.5.2 Multi-thread Checkpointing

As an (Jackal) application consists of multiple threads per process, the checkpointing algorithm has to checkpoint all application-level threads of the process. If JaMP is enabled for the application, there also exist OpenMP/Java threads that form the current thread team. In addition, there are Jackal-specific service threads (such as the garbage collector, the finalizer thread, and communication handlers) that are needed by the Java runtime environment [78]. Since the service threads are known to the runtime system, no checkpointing is necessary. The service threads are inactive most of the time and do not own any computational state that has to be conserved. The runtime system can restart these threads from scratch upon restart.
Application-level threads (i.e., OpenMP/Java threads and other programmer-defined threads) are subject to the checkpointing algorithm. For each active thread of the process, the call stack has to be unwound and saved by applying the check pointers for each stack frame.

To create a consistent checkpoint, no thread may continue computation while another thread is being checkpointed. Otherwise, race conditions occur that may cause lost updates. A lost update occurs when the checkpointed thread is reading data on the heap and another thread is concurrently modifying the same data. In this case, the update might not be reflected in the checkpoint and the computational state contained therein is no longer valid. We discuss the creation of the heap checkpoint in Section 5.5.3.

There are two potential solutions to solve this problem. First, each access to the heap could be protected by a lock used by both the application and the checkpointing runtime. The lock would ensure mutual exclusion between normal operation of the application threads and the checkpointing runtime that writes data to a checkpoint. This solution, however, deteriorates performance, since the lock becomes the bottleneck even for small numbers of threads, limiting concurrency of disjoint heap accesses.

Second, the threads can be collected in a special checkpoint barrier that guarantees that no thread proceeds during a checkpoint and that checkpointing only starts if all threads are blocked. This solution permits regular heap accesses to be performed without locking. Moreover, overhead is only caused when a checkpoint is created. Due to this advantage, we decided to use this solution for our checkpointing algorithm.

Since the compiler adds the `createCheckpoint` automatically, all threads will eventually encounter a call to `createCheckpoint`. The threads enter the checkpoint barrier and wait for the arrival of the other threads. When all threads have reached the barrier, the runtime system selects one thread and checkpoints its call stack. After the checkpoint has been created, the thread is put back into its waiting state. All other threads are processed the same way. Threads may enter a synchronization operation (e.g. `Object.wait()`, `Thread.sleep()`) before they encounter the `createCheckpoint` call. In this case, the synchronization primitive tests for an active checkpoint and if needed they also proceed to the checkpoint barrier.

Threads that are already blocked in one of the synchronization methods need a special treatment. These threads are potentially blocked for a long
period (or may remain blocked forever if the notifying thread is already collected in the checkpoint barrier). Hence, the checkpoint would be deferred until all blocked threads have left their synchronization primitives. As this is undesired and threads should be checkpointed as soon as possible, our solution is to wake up the threads. The checkpointing runtime sends a signal to all blocked threads by throwing a `CheckpointException`. The exception is caught in the synchronization primitive and the exception handler calls `createCheckpoint`. After the computational state has been saved, the threads re-enter the synchronization primitive and are thus sent back to their blocking state. From an application point of view, these threads seem to have never left the synchronization primitive.

### 5.5.3 Checkpointing the Static Data and the Heap

Using the aforementioned stack-walk algorithm and the checkpointing barrier, it is now possible to save the computational state of all threads in a single process. As the heap contains a large fraction of the application’s state stored in dynamically allocated objects, the objects on the heap have to be stored as well. It is sufficient to only store objects that may contribute to future computation. In addition, the `static` fields of a class need to be checkpointed.

As the compiler knows about all classes that have been compiled, it can create functions that read all static fields of the class and store them into the checkpoint. In addition to the values of the static fields, meta information has to be written to disk. The meta information records whether or not the static initializer of the class was already executed. This is important, as the JLS prescribes that upon first access to a class, the static initializers of the accessed class have to be executed to cleanly initialize the static fields. If the application is restored, the already executed static initializers may not be re-executed. When the application resumes from the checkpoint, the static fields are loaded and the checkpointing runtime disables the static initializers that have already been executed.

Java objects are accessed by means of references [78], which can be implemented by pointers into the application’s heap. However, the value of a pointer is only valid if the same memory layout is used on all architectures. Reality shows that different architectures have different requirements on, for example, the alignment of in-memory data. Usually, 32 bit machines require
an alignment at 4-byte boundaries, whereas 64 bit machines mostly enforce an 8-byte alignment. Hence, the internal memory layout of individual objects generally varies between architectures. Fields of objects (potentially) have to obey the same alignment rules as their surrounding object. Furthermore, on a 32 bit architecture a pointer field is 32 bits wide, while on 64-bit architectures the pointer needs 64 bits. Finally, in general, the endianess of the individual fields in the object differs.

For writing the application’s heap to the checkpoint, each object that resides in the heap must be converted to a generic format and written to disk. As Java’s object serialization facility [184] already defines a platform-independent format for storing objects on disk, we reuse this technique to checkpoint objects. Object serialization transitively follows an object reference that is contained in the currently serialized object. Hence, the complete graph is automatically serialized to disk.

In contrast to Java’s serialization, our checkpointing solution does not directly write the checkpointed objects to disk. Instead, they are handed off to a service thread for compression. Although it is more expensive to compress the checkpoint, the advantage is that the created checkpoint consumes less disk space. Moreover, the additional overhead is compensated, as the checkpoint may be shipped faster over low-bandwidth WAN connections.

As already mentioned, only objects that contribute to the future computation need to be stored during checkpointing. Hence, the checkpointing algorithm has to identify these objects, which is impossible to do programmatically (due to the halting problem). A similar problem is found in garbage collectors [108]. Garbage collectors need to determine the same kind of objects to be able remove the superfluous objects from the heap.

Garbage collectors identify a so-called root set of objects that are directly reachable by the application’s threads. Objects referenced in the root set and its reflexive transitive closure, need to be kept in memory as they might be accessed in the future. All other objects may be discarded. Our checkpointing solution relies on the same technique. Using the root set of accessible objects, Java serialization is applied to each object referenced by the root set. If the root set of a thread is completely processed, all objects accessible by this particular thread have been saved to disk. Since objects may be shared between two threads and objects should be serialized only once, the hash table used for cycle detection is re-used to detect objects that already have been serialized during the checkpointing process of another thread.
All local variables, method arguments, and static fields accessible to the methods on a thread’s call stack constitute the root set. During the stack-walk that checkpoints the live variables in each stack frame, data that might reference an object in the heap is added to the root set of the thread. At this stage, no difference is made between, for example, an int variable that by chance contains a value equal to a valid heap address and an Object variable. From the checkpointing point of view, a too large root set only causes checkpointing of superfluous objects.

During restoration of the application heap, the serialized objects are deserialized into the data segment. Deserialization is the inverse operation to serialization and reads a byte stream from disk and recreates the object graph in the heap. While deserialization does not guarantee the relative ordering of objects in the heap to remain stable, all objects in the object graph are still connected the same way as before the serialization was performed. Please note that creating a checkpoint and restoring the application from a checkpoint comprises an implicit garbage collection, as only the reachable objects are actually stored in the checkpoint and restored from there.

## 5.6 Distributed Checkpointing Algorithm

A Jackal application consists of a set of cooperating processes that together form the application. As each process potentially is located at a different node of the cluster, processes are loosely coupled with the other processes, that is, there exists no memory area that is shared among two processes (also known as shared nothing). Processes communicate through the inter-connection network of the cluster by means of typed messages. Almost all Jackal communication is performed in a ping-pong fashion. If one process sends a message to one of the others, most of the time a reply message is expected that either contains the requested data or an acknowledgment.

As has been discussed above, a single process can collect all of its threads in a checkpoint barrier to ensure that no concurrent modifications to the computational state occur while the checkpoint is created. However, a naive extension of this solution to a distributed environment, that is, to simply run the single-process checkpointing algorithm for each of the cooperating processes, does not yield a valid collective checkpoint.
This section presents the issues of distributed checkpointing algorithms (Section 5.6.1) and introduces a distributed checkpointing algorithm that does not suffer from the aforementioned correctness problem (Section 5.6.2).

5.6.1 Introduction to Distributed Checkpointing

The issue of creating a combined checkpoint in a distributed environment of cooperating processes is twofold.

First, a distributed system does not have a global clock. Each subsystem (i.e., cluster node) maintains its own local clock. Although the clocks can be synchronized by, for example, the NTP protocol [139], there is always a time difference of a few microseconds [140]. In this period of time, a processor executes millions of instructions that may concurrently modify the computational state of the process while the state of another process is already frozen and stored in a checkpoint. This scenario easily leads to an inconsistent global checkpoint.

Second, the computational state of a multi-process application not only consists of the computational states of all the individual processes, but the state of the network also contributes to the overall state. If a message was delivered to the network at the sender side, but not yet received at the receiver side, the in-transit message belongs to the computational state of the combined system and has to be checkpointed. During restoration, the message has to be restored on the network to ensure that the receiver eventually receives this particular message. Alas, the state of the network is invisible to the individual processes, that is, the contents of messages are inaccessible while in transit.

The inaccessible network causes two problems that are known as the orphan messages problem and the lost messages problem [103].

A message is said to be an orphan message, if the send event is not recorded in a checkpoint, but the receive event has been recorded. Figure 5.7 shows two processes that create a checkpoint. Process P1 creates the checkpoint before it sends out a message to process P2. Process P2 receives the message and then creates its local checkpoint. This situation leads to a combined checkpoint in which the send event for the message is missing, while the receive event has been recorded. When the application is restored and process P1
sends the message again, process P2 receives the message twice and proceeds with an invalid computational state.

Lost messages emerge if the scenario is reversed (see Figure 5.8). Process P1 now records in its local checkpoint that a message has been sent. Process P2 creates its checkpoint before it received the in-transit message. Thus, the combined checkpoint contains a message that has been sent, but has not been received by the target process. For the same reasons as with orphan messages, lost messages also cause an incorrect computational state upon restoration of the application.

A global checkpoint consisting of several local checkpoints is said to be consistent if all of the following three rules are fulfilled.

1. The computational state of each process is reflected in at least one checkpoint.
2. No orphan messages exist in the global checkpoint.
3. No lost messages exist during checkpoint creation.

Otherwise the global checkpoint is called *inconsistent*. In Figure 5.9, the *recovery line* (indicated by a dashed gray line) depicts a consistent set of checkpoints that can be used to store the processes P1 to P3 into a consistent computational state. Of course, for our reparallelization and migration framework, the distributed checkpointing algorithm must always create consistent checkpoints.

### 5.6.2 Distributed Checkpointing Algorithm

There are two main categories of distributed checkpointing algorithms that provide a means to create a consistent global checkpoint [110].

First, in an independent checkpointing algorithm, the processes individually create local checkpoints in regular intervals. During normal operation, each process maintains a log of sent and received messages. At the time the processes are restored from the checkpoint, the algorithm tries to find a recovery line of consistent local checkpoints that obey the above rules, starting at the most recent checkpoints. If rule 2 is violated by a message in the logs, the sender has to rollback to an earlier checkpoint. If rule 3 is broken, the receiver has to rollback to an earlier checkpoint. This is repeated until
5.6 Distributed Checkpointing Algorithm

![Diagram of distributed checkpointing algorithm]

(a) In-transit message between node $M$ and $A$ (dashed arrow).
(b) Node $M$ broadcasts a control message (solid arrows).
(c) The nodes $A$ and $B$ also send a broadcast.
(d) The control messages have emptied the network.

Figure 5.10: Visualization of the distributed checkpointing algorithm.
the above rules are all fulfilled. This kind of algorithms is vulnerable to the so-called domino effect that causes continuous rollbacks until a consistent recovery line has been found. In the worst case, the only consistent recovery line is located at application start-up. For obvious reasons, it is not an option to restart a migrated application from scratch if all checkpoints are inconsistent.

Second, coordinated algorithms enforce a consistent recovery line when creating the checkpoint. Hence, repeated rollbacks and the domino effect are avoided by design. The most simple solution to achieve a consistent recovery line at checkpoint creation, is to freeze every process and to empty the network. Cleaning the network is trivial if in-transit messages on the network are subject to FIFO communication. With FIFO communication, messages cannot bypass each other if they are sent by the same sender and are directed to the same receiver.

The checkpointing algorithm can send a control message from one process to all other processes to push the in-transit data messages (see Figure 5.10(a), dashed arrow) through the network to their respective receivers (Figure 5.10(b), solid arrows). If a node receives a control message, it stops delivering messages and also sends a broadcast to push all messages that are on the network (Figure 5.10(c)). After all processes have received a broadcast from all other processes (Figure 5.10(d)), the network cannot contain any data messages, if the processes did not send any data messages after they have received the control message. This algorithm needs $O(p^2)$ messages, where $p$ is the number of processes participating in the algorithm.

As our solution creates only a single checkpoint during the migration of an application, (repeated) rollback actions have to be avoided under any circumstance. We have decided to implement a coordinated algorithm, as it guarantees that no rollback action is necessary at application restart. The quadratic growth of message counts with increasing node numbers can be tolerated: measurements have shown an algorithmic overhead of roughly thirteen milliseconds for sixteen processes interconnected with Gigabit Ethernet. Compared to the time needed to create the checkpoint of the application and write it to disk, this overhead is negligible. Moreover, it has been shown that coordinated checkpointing is the best choice for fast network interconnects [48, 174].

To implement the coordinated algorithm, each Jackal process executes an additional service thread, the CheckpointManager. The CheckpointManager
on the process with rank 0 is considered the master thread, whereas the checkpointing threads on the other processes play the role of slaves. The runtime impact of the additional checkpointing threads is negligible, as the threads are sleeping in blocked state most of the time. On an external event, that is, a checkpoint request, the master wakes up and initiates the algorithm. The slave threads wait for messages sent by the master or other slaves and only participate in the algorithm, if the checkpoint algorithm is actively executing.

Our algorithm is based on four kinds of control messages: \textit{init}, \textit{ack}, \textit{abort}, and \textit{start}. The \textit{init} message is the primary control message to inform all slaves about the checkpoint request. With \textit{ack}, the slaves report to the master that they are ready for taking a checkpoint. If a slave is not ready for checkpointing, it responds with the \textit{abort} message. Having received an \textit{ack} message from all slaves, the master sends out \textit{start} messages to all processes indicating that the local checkpoints can be created, since the network is cleared from in-transit data messages.

Figure 5.11 shows an example execution of our coordinated checkpointing algorithm. The master process M initiates the algorithm by sending an \textit{init} message to all other processes. Process P2, upon receiving the \textit{init} message,
sets up the local checkpointing barrier and waits for the threads to block. As soon as all local threads are collected in the barrier, process P2 broadcasts the \textit{init} message. Before receiving the broadcast from M, process P1 sends a data message (e.g. an object request message) to P2. If P2 receives this request message, there are two possible strategies on how to handle the incoming message. First, the process P2 can put aside the message and handle it after the checkpoint has been created. Second, the process can handle the message immediately and send the reply message to the requester.

The first solution is problematic, since saving the message for later handling, is a complex task, as parts of the message might still be in-transit on the network. In addition, saving the message buffers of Jackal is not possible, since the buffer handling is located in an external communication library and hidden behind the OS networking API, which cannot be compiled with the Jackal compiler and is thus not prepared for the heterogeneous checkpointing solution proposed in Section 5.4. For the same reason, the service threads that handle incoming message cannot be checkpointed.

We have decided to permit the communication threads to immediately handle incoming messages. P2 receives the incoming message, hands it over to the service thread for handling it, and sends out a reply message to the requester (indicated by a circle in Figure 5.11). This, however, breaks P2’s promise to not send any messages, after it has received the \textit{init} message. Our algorithm handles this case by means of the \textit{abort} message. If a process P has received the broadcast from all other processes, it checks the communication system whether or not a message has been sent after it received the first broadcast (in the example, P2 sends a reply message). If no message was sent, it sends an \textit{ack} message to the master. Otherwise, an \textit{abort} is delivered to the master. The master proceeds to the next checkpointing phase, as soon as all other processes have responded with a \textit{ack} message. In any other case, the master restarts the checkpointing algorithm by sending out another wave of \textit{init} messages (see phase 2 in Figure 5.11).

The algorithm is guaranteed to terminate after at most two phases. Since each process sets up a local barrier when it receives the \textit{init} message, only processes that did not yet receive the \textit{init} can cause an abortion of the algorithm. Threads that still proceed with computation and cause messages on the network can exist only on these processes. Processes that already received the \textit{init} message set up the barrier and collect all local threads in the barrier. Hence, these threads do not cause any messages on the network.
5.7 Checkpointing the DSM Protocol State

The algorithm enters the second phase only due to an abortion. Because all processes have received at least one *init* during the last phase, the barriers are active and the threads are still blocked. Thus, no further violations may occur in the second phase, because no thread proceeds in computation and no data messages are sent through the network.

A similar algorithm is executed when the application resumes from a checkpoint. Each process is restarted individually from its local checkpoint. To avoid race conditions during the restart, the local checkpointing barrier is not torn down directly after the restoration of the local process was completed. The process sends out a broadcast, indicating that restoration was successful. Only after each process has received this message from all other processes, the local checkpointing barriers are torn down and execution proceeds as normal.

5.7 Checkpointing the DSM Protocol State

Jackal’s DSM hides the individual nodes’ boundaries by simulating a globally shared address space on top of the nodes’ distributed local memories. Jackal respects the JMM [135] and thus has to maintain coherence of the shared memory. The component of Jackal that ensures coherency and exchanges data is the DSM protocol of Jackal’s runtime system. Obviously, the DSM protocol is message-based. Hence, the distributed checkpointing algorithm presented in Section 5.6 is sufficient to ensure that no messages of the DSM protocol are orphaned or lost during checkpoint creation.

To provide coherence to the application, Jackal needs to store meta data that describes the state a particular data object is in. This meta information has to be checkpointed as well. If the information was lost during checkpoint creation, Jackal would not be able to correctly execute the application after a restart. This section investigates the internal states of the DSM coherency protocol (Section 5.7.1) and describes how they can be saved to a checkpoint and restored from there (Section 5.7.2).

5.7.1 The Jackal DSM Protocol

Before we can present how a DSM-coherent checkpoint has to be created, it is important to understand the internal states of Jackal’s DSM protocol.
Section 3.2 has already briefly introduced the Jackal DSM. We will now take a closer look on the internals of the DSM protocol.

Jackal implements an object-based DSM [203], that is, Java objects are used as the granularity of data transfers and memory consistency. To simulate the global address space, objects are transferred over the network and cached on the accessing nodes. To implement the object cache, Jackal subdivides the heap of each process into three distinct areas: *allocation heap*, *caching heap*, and *administrative heap* (see Figure 5.12). In the allocation heap objects reside that have been created on the local node. The caching heap stores objects that have been allocated on a remote node, but are used on the local node. The DSM protocol ensures that the copies of objects located in the caching heap are kept consistent with their respective primary copy. The administrative heap is internally used by Jackal to maintain state information for the DSM protocol.

The Jackal compiler prepares the executable for execution in the DSM environment by prefixing each access to an object with a so-called *access check*. Figure 5.13(a) shows a short Java code example that reads a field from an object. The added access check is depicted in Figure 5.13(b) as Java-like pseudo-code. The access check tests if the object is already available in
5.7 Checkpointing the DSM Protocol State

(a) Example code before compilation.

```java
class AccessCheckExample {
    int foo(SomeObject o) {
        return o.field;
    }
}
```

(b) Example code after insertion of the access check (pseudo-code).

```java
class AccessCheckExample {
    int foo(SomeObject o) {
        if (!readable(o))
            fetch(o, readable);
        return o.field;
    }
}
```

Figure 5.13: A function before (Figure 5.13(a)) and after the insertion of an access check (Figure 5.13(b)).

the right access mode on the current node. If the accessed object is not available, the check triggers the Jackal DSM runtime system and sends a message to the node (the **home node**) that stores the primary copy of the object. This request is answered by a message that contains the requested object’s data.

The access check test is implemented using per-thread read/write bitmaps in the administrative heap of the local node. For each access mode (i.e., read or read/write), the DSM runtime maintains a bit list with one bit for each object on the local node. If the bit is set, the object is already available. After the object has been transferred to the local node, the DSM runtime system sets the bit to reflect the state change for the requested object. On the home node, the DSM runtime also maintains a per-object bitmap (the **requester bitmap**) that stores information about the nodes that currently own a copy of an object. The bitmap contains as many bits as there are nodes and a bit is set if the corresponding node has requested an object and did not yet flush the object’s copy back to the home node.

When a flush operation occurs, threads are forced by the JMM to write back their private caches to the shared memory area. When a thread enters a monitor object, that is, it enters a synchronized method or block, the
thread has to flush its data cache to retrieve updated values from shared memory and to write back modified data. Jackal implements flush operations by writing back modified data in objects to their respective home nodes (see below) and by resetting the corresponding bits in the read/write bitmaps.

To uniquely identify each object that resides in the DSM address space, a so-called Global Object Reference (GOR) is created for objects. A GOR is a tuple of the form \((\text{alloc-node}, \text{alloc-address})\), where \(\text{alloc-node}\) is the logical rank of the node that created the object and \(\text{alloc-address}\) is the object’s address in the allocation heap on that node. The GOR is fixed during an object’s life time; only if the object is reclaimed by the garbage collector, the GOR is released and may be recycled.

When a non-local object is received, the runtime system assigns it a memory location in the caching heap of the local node (see Figure 5.12). This ensures that subsequent object accesses can be performed locally without having to frequently access the object though its GOR. For status updates and data updates, the GOR must be accessed to retrieve the home node of the object. For performance reasons, Jackal stores the GOR directly in the object header that is also used for storing the object’s type and virtual method table [16].

Any writable object that resides in the caching heap is assigned a shadow copy (the twin) of the original object data received during the access check. The twin is used to track modifications to objects and to create a differential object image when a modified object has to be flushed to its home node. The runtime system only sends back the differential image to the home node and applies it to the primary copy.

### 5.7.2 DSM State Checkpointing

While not being directly accessible by the application, the DSM protocol state presented in the last section constitutes an important part of the application’s computational state. Let us now investigate which parts of the DSM state and the heaps have to be checkpointed to reflect the complete computational state in the global checkpoint. Finally, we present how our system conserves information about Java locks that the threads currently hold.

The discussion in Section 5.5.3 revealed that the objects residing in the application heap constitute a large fraction of the computational state of
5.7 Checkpointing the DSM Protocol State

an application. In the DSM environment this central heap of each Jackal process is subdivided into three distinct heaps with different semantics. The allocation heap is primarily used to allocate objects and thus has to be entirely stored in the checkpoint.

If an object is restored from the checkpoint into the allocation heap, the object’s address in the allocation heap changes. If the architecture of the machine changes during migration, the position of the allocation heap in the machines address space is likely to differ. Furthermore, the restart of an application automatically compacts [108] the allocation heap to avoid fragmentation of the allocation heap due to objects that have been unreachable during checkpointing and, thus, are not included in the checkpoint.

Changing an object’s address in the allocation heap breaks the GOR that is associated with the object. During restoration from a checkpoint, the GOR has to be updated to reflect the new storage location of the object behind the GOR (the rank information is kept fixed for now). As other nodes also reference objects by means of the GOR, the GORs used by remote nodes have to be updated as well. During restoration, the old and the new GOR have to be associated with each other by a mapping table. After all local objects have been restored to the allocation heap (and the GORs have been updated) a node sends a broadcast and each node starts to send out messages that ask about updates to GORs of cached objects. As the rank information in the GOR did not change during restoration and as each node knows about the objects residing in its local caching heap, the request for update is only directed to selected nodes.

Although the caching heap only stores copies of objects, they still need to be checkpointed, since the writable copies of objects in the caching heap might have been modified by one of the application threads. These updates have to be reflected in the checkpoint, as otherwise the application would be in an inconsistent state after restart. To checkpoint these objects, two solutions are possible.

The first solution stores the copies of objects in the caching heap in the checkpoint. Additionally, the home nodes of these objects store the primary copy in their local checkpoint. Thus, a single object might be included in several local checkpoints and may have different internal states in each checkpoint. At restart, the primary copy and the copies are restored on their respective nodes. Each copy in the caching heap has to be re-associated with the corresponding primary copy to make an object’s state consistent.
5 Heterogeneous Distributed Checkpointing

The second solution flushes each modified object on the caching heap to its home node before checkpointing. In this case, the caching heap is emptied and the primary copy contains the most recent state of the objects. The JMM [135] permits such a flushing behavior during checkpointing and correct Java applications should not suffer from semantic errors. While being easy to implement, data locality is lost, as the remote nodes’ caches are empty after restart. Hence, the application suffers from a performance penalty until the Jackal DSM has re-established data locality. Nevertheless, we use this solution for our checkpointing implementation and tolerate the temporary performance loss, as the Jackal DSM quickly adapts to the locality needs of the application.

Since the caching heap is emptied by a flush operation before the checkpoint is created, the read/write bitmaps and the requester bitmaps do not need to be stored to the checkpoint.

Information about locks that a thread currently holds is checkpointed by writing the lock owner into the checkpoint. In most cases, the IDs of threads are platform-dependent; the thread ID is either a pointer to a thread-information data structure or a process ID and thus changes during restart. Similar to updating GORs, the new and the old thread ID are stored into a mapping table. When the lock information is restored in the restarted application, the mapping table is used to replace the old ID of the lock owner with its new ID.

5.7.3 Changing the Number of Processes during Restoration

Using checkpointing for migrating an application between different systems of a computational Grid requires that the number of nodes can be changed during restart of the application. The target system might offer more cluster nodes to the application or might require a smaller degree of parallelism. Our checkpointing solution supports both merging local checkpoints into smaller numbers of process images and creating empty process images.

If the number of processes has to be increased, the new nodes start with an empty image (see Figure 5.14). Empty images only contain Jackal’s service threads and provide empty heaps for object allocations. The caching heap and the administrative heap are empty as well and are only initialized for future use. The example of Figure 5.14 shows three processes P1–P3 that
5.7 Checkpointing the DSM Protocol State

are restored in the processes PA–PC. If new threads are spawned by the application, the Jackal runtime system automatically places the new threads on the idle processes (PD–PF in the example). Adapting data placement to the application’s needs, the heaps of the empty process images start to fill with allocated and cached objects.

For a restart of the application on a smaller number of nodes, the checkpointed process images have to be merged into a smaller number of process images (see Figure 5.15). In this case, the restoration algorithm expects information about which local checkpoints have to be merged into a single process image. According to the mapping specified, checkpoints are loaded and are sequentially placed into the process images. In the example of Figure 5.15, the processes P1–P6 are merged into the processes PA–PC. The application heap of the old processes are merged into the heaps of the new process set. The caching heap and the administrative heap are not merged, as these heaps are empty at checkpointing time.

In each data structure that references the old rank of a process, the number has to be updated to match the new rank of the resulting process image. GORs are affected by the rank changes as well; the rank in the GOR needs to be updated. When retrieving address updates to the GOR, the requester
5 Heterogeneous Distributed Checkpointing

Figure 5.15: Merging six process images into three process images.

directs the message to the new process rank and ships along the old rank. This is necessary to provide information to the target node to decide from which process image the GORs in the request message are.

When the caching heaps that stem from different process images are merged into a single target process image, the caching heap may only contain one copy of the object along with a single twin of the object. As the caching heaps of the processes have been flushed before the checkpoint was created, no cached (modified) copies or twins exist in the caches. Hence, this problem is solved automatically during checkpointing and no further actions become necessary at restart.

5.7.4 Interaction with Reparallelization

The reparallelization runtime (see Section 4.7) and the checkpointing runtime need to cooperate for application migration. When the application is resumed from a checkpoint and the number of processes changes, the reparallelization runtime has to be informed about this change. The reparallelization runtime then may choose to adapt the degree of parallelism of the currently active parallel region as necessary.
5.7 Checkpointing the DSM Protocol State

Although both components have to interact, both provide orthogonal features to the Jackal runtime system. Hence, their implementation should be kept as independent as possible to not limit the independent usage of both components. The checkpointing runtime should be fully functional in an application that does not exploit reparallelization or even OpenMP/Java. Conversely, OpenMP/Java’s reparallelization runtime shall be kept functional without the checkpointing runtime enabled.

The observer pattern [74] satisfies our requirement of separation of the components while providing a means for the components to interact. The checkpointing runtime offers interested components of the Jackal runtime system two kinds of observable events: \texttt{CheckpointObservableCheckpoint} and \texttt{CheckpointObservableRestore}. Both inherit from Java’s standard observable implementation \texttt{java.util.Observable} and provide the required loose coupling of the checkpointing runtime and the reparallelization runtime.

When the checkpointing algorithm has finished creating the checkpoint, the \texttt{CheckpointObservableCheckpoint} event is triggered and the observers are notified. This, for example, can be exploited by the migration framework (see Chapter 6) to receive a notification as soon as the application’s computational state is stored into the checkpoint. It can then safely terminate the execution of the application and transfer the checkpoint data to the target system.

With the \texttt{CheckpointObservableRestore} event it is possible to monitor the restoration process of an application. The observer attached to this event receives a notification after the application was restored by the checkpointing runtime. Along with the notification, the process mapping table is passed. The mapping associates the old process ranks with the new rank of a process, if a process was merged with another process image at application restart. The mapping can be read by the reparallelization runtime to remove threads or to create new threads as necessary.

The reparallelization runtime needs to adapt the application’s degree of parallelism to the new number of available PEs after restart (see Section 4.2.2). Changes in the computing environment have to be detected such that the reparallelization runtime can determine the correct thread count. In addition, the reparallelization runtime needs information about merged processes to improve data locality. After reparallelization, a thread should be close to its former process image to avoid excessive data migration. The reparallelization runtime’s default strategy object (see Section 4.7.1) registers itself
at the CheckpointObservableRestore. After a restart, the strategy object receives the restart notification and reads the new process mapping. It then adapts the application’s degree of parallelism as needed and places threads close to their former process image.

5.8 Impact of Checkpointing on Application Semantics

Checkpointing of a Java application may not interfere with the language specification set up by the JLS [78]. If checkpointing the application violated the semantics of Java, transparent migration of Java applications would be impossible. In this case, the effects of a migration would be visible to the programmer. Moreover, the programmer would be forced to modify the application’s code to handle the effect and to recover from the semantical violations.

This section covers all issues related to checkpointing of Java applications and argues about their conformance to the JLS. Whenever checkpointing would break with the semantics of the Java programming model, we present a solution how to avoid the violation.

An application is stopped by collecting all local threads of a process in a checkpointing barrier while a checkpoint is created. The distributed checkpointing algorithm runs to clear the network from in-transit data messages. If the application requests the current time at two program locations (e.g. by means of System.currentTimeMillis()) and a checkpoint is taken between these locations, the program notices an extra delay compared to the execution without checkpointing. Fortunately, the JLS does not make any assumptions on the processing speed and thus such delays do not cause any semantic problems. Similar delays can be observed in JVMs or Jackal when the garbage collector discards unused objects in the application heap.

Another timing issue stems from the checkpoint/restore process. After the application’s state has been stored in the checkpoint, the application may be restored after an arbitrary period of time. From the application’s point of view, time is no longer continuous, as the application is not aware of being frozen in the checkpoint. If restored from the checkpoint, time seems to jump to a future point in time or the clock might move backwards if the application is moved between time zones. While the non-continuous time line is tolerated by the JLS, it might cause problems for applications that
are sensitive to timing issues. Except for real-time applications, we are not aware of any Java application that might suffer from semantic problems. If the application is sensitive to timing, the only solution relies on the observer API of the checkpointing package to receive a notification about the checkpointing. The application can determine the current time before the checkpoint has been created and after the application was restored. With this information, the application could compute the time difference and may arrange for a potential time offset and act accordingly.

The checkpointing runtime system prematurely wakes up threads that are blocked in synchronization primitives and cannot reach `createCheckpoint` (see Section 5.5.2). The checkpointing runtime throws an exception to wake up the thread and the exception is caught in the synchronization primitive. After the thread was checkpointed, it is sent back to its blocking state. This complies to the JLS, as the thread does not leave the synchronization primitive during checkpointing. However, `Thread.sleep()` may cause problems, since the delay for taking the checkpoint might exceed the delay specified for `Thread.sleep()`. In this case, the synchronization primitive seems to sleep longer than expected. As `Thread.sleep()` only ensures that execution is delayed for at least the specified amount of time [147], longer delays do not cause semantic problems.

An invocation of `hashCode()` has to return the same value during an object’s whole life time [187]. If the method is not overwritten by an object’s class, the runtime system uses the object’s address in the machine’s address space. This is feasible as long as the execution environment does not change the object’s position in the heap. When restoring the application’s heap from the checkpoint, the position of objects in the machine’s memory is likely to change. Computing the return value of `hashCode()` on the fly by returning the object’s current address would break the contract for the `hashCode()` method. We solve this issue by applying memoization to the object’s first address that was assigned during the allocation of the object. From there on, `hashCode()` consistently returns the stored address as the object’s hash value.

Finally, changing the number of nodes that execute the application may also cause semantic problems. Section 4.6 has already shown semantic problems that arise from changing the number of threads during program execution. As Jackal’s API exposes methods to query the number of processes that execute the application, the application might retrieve this information and
store it in variables. This, however, is not permitted, as after restoration the application could continue execution with invalid process counts or process ranks. The application programmer has to be aware of this issue and design the application accordingly. For example, the application might register itself at the observer API of the checkpointing runtime to be notified about changes in the process set.

5.9 Summary

In this chapter, we have presented a technique to migrate applications in Grid environments. We have analyzed the needs of checkpointing for fault-tolerance and deduced requirements for a migration solution. Only with these requirements, an effective means to transport an application’s state between different systems in a Grid environment is possible.

To enable migration for an application, the programmer should not modify the application; our compiler adds the migration feature transparently. The overhead of the migration is kept as small as possible during normal operation of the application. The checkpointing package creates compressed checkpoints, since a transfer over low-bandwidth WAN connections is needed. Our solution does not extend the kernel, but augments the application with the checkpointing runtime. Finally, our solution stores the physical representation of the application state in a platform-independent checkpoint.

We have discussed the need for a platform-dependent checkpointing solution. We have also analyzed the computational state of an application from a system programming and compiler-centric point of view. We identified the constituents of the computational state and investigated which fractions of the application state are platform-dependent. We have considered both the state stored in the operating system as well as the state contained in the application process.

For each of the constituents, we propose low-overhead checkpointing solutions. To create platform-dependent checkpoints of a single thread, we have introduced the notion of unique descriptor strings that have already been presented in [204] for thread migration purposes. The UDS is used to platform-independently describe the storage location of a live variable.
Starting with checkpointing of single threads, we have incrementally extended our checkpointing solution to conserve the state of a single-process with multiple threads.

The multi-process checkpointing algorithm is based on a coordinated stop-the-world algorithm. The algorithm uses broadcasts of control messages to empty the network and to avoid inconsistent checkpoints caused by orphaned and lost messages. Clearing the network from in-transit messages effectively solves the consistency problem. We described how our checkpointing algorithm stores the state of Jackal’s DSM protocol. Upon restart, our system can merge process images or launch empty process images to adapt the application to the available number of processing elements.

Finally, this chapter has discussed the impacts of checkpointing to application semantics. We have identified issues that potentially cause semantic problems. Our checkpointing solution fully respects the requirements of the JLS. Where necessary, our checkpointing package provides efficient solutions to these semantic problems.
6 Framework for Inter-cluster Migration

In the previous section, we presented a checkpointing technique that suits the needs of application migration. The checkpointing system freezes the application’s state and stores it into a checkpoint on stable storage. From the frozen state, the application can later resume and continue computation at the point at which the checkpoint has been taken. Our system creates platform-independent checkpoints to transfer the application to a remote system such that it can restart execution there. Our checkpointing technique is completely transparent to the programmer and only imposes a negligible overhead on the application’s runtime.

This chapter presents a framework that automates the process of migration. If an application is about to exceed the local reservation, the it is instructed to create a checkpoint. The framework selects a target system for the migration and transfers the saved state to the target. Finally, the framework restarts the application from the transferred checkpoint.

The framework is called *Open Grid Research Environment* (OGRE) and is intended to provide a flexible framework architecture for future research on migratable applications in Grid environments [119]. We present a prototype of the framework and describe its architecture. To prove the feasibility of the framework, we implement a basic resource discovery module and a simple migration strategy that migrates a single application between clusters. Our migration strategy is based on auctions, that is, each accessible cluster reports a bid and an auctioneer selects the most promising bid according to the migration strategy.

Before the chapter explores the implementation of the OGRE framework, the state of the art in Grid runtime environments is presented (Section 6.1).
Section 6.2 presents the design objectives of the OGRE framework. In Section 6.3, we describe the component architecture of our migration framework and the interaction between the components. Section 6.4 focuses on the migration strategy. Finally, the technique to transfer the application state to the migration target is shown in Section 6.5.

### 6.1 State of the Art

Existing frameworks that provide a middleware layer for Grid applications target production-stable environments. Our framework, however, is intended to be a research framework that can be used for research on application migration on top of the production-stable environments. Our design objectives (see Section 6.2) focus on a high degree of flexibility.

The Open Grid Services Infrastructure (OGSI) [71] is an attempt to define a common interface to Grid services, that is, functions that are exposed by the Grid resources. As OGSI is built on top of Web services [156], implementations have to provide a complex infrastructure of dedicated systems that expose the services. OGRE is explicitly designed to avoid the need for such complex infrastructures.

The Globus Toolkit 4 [70] is a well-known implementation of OGSI [71]. Also for UNICORE [166] an OGSI integration exists. Both provide components for security, authentification and authorization, data transfer, resource discovery and allocation, and Grid-enabled MPI. As they offer a powerful middleware layer that closely interacts with the cluster management system, both must be installed and deployed on the cluster systems. Thus, users cannot rely on an installation on all accessible systems. OGRE ships all functionality needed in a single Java executable and can be used on any cluster for which a JVM is available. In addition, OGRE is explicitly designed for application migration and may cooperate with the Globus Toolkit and UNICORE.

The earlier mentioned projects Cactus [12, 76], DGET [63], GrADS [200], and P-GRADE [121] also offer middleware services for Grid applications and target migratable applications.

Cactus and DGET violate our requirement of supporting any programming language and any programming model, as they enforce programmers to ex-
tend the framework with application-specific functionality. OGRE supports any programming model and programming language by a thin communication layer that performs communication between the migration framework and the application.

GrADS and P-GRADE provide a framework for the execution of MPI applications on computational Grids. Focusing on MPI applications, GrADS and P-GRADE cannot migrate applications that are implemented in other programming models.

All of the above mentioned projects solve the resource discovery and resource selection problem either by providing their own schedulers or by directly interfacing with existing cluster management systems. OGRE implements resource discovery by accessing the existing cluster scheduler. Instead of a tight integration through API calls, we keep our system flexible and retrieve the needed information from various sources in a cluster by means of a batch system communicator. Whereas the above projects ship with a fixed implementation of the resource selector, our framework contains a resource selection component that can be easily replaced with other components.

6.2 Design Objectives

The design of the OGRE framework is governed by the aim to provide a foundation for research on migratable applications in Grid environments. As it is a prototype implementation, the requirements are different from a production system. We will now explore the design objectives and describe how they influence the architecture of the OGRE framework.

Being a research prototype, OGRE has to be as flexible as possible to enable researchers to modify and adapt the behavior of OGRE as needed. Hence, OGRE cannot be implemented as a monolithic system that performs a predetermined control flow when an application requests a migration to a target system. Instead, OGRE has to be decomposed into components that together form the framework. Each of the components is responsible for implementing a certain aspect of the migration steps. To be flexible, the OGRE components can be replaced by components of the same type (i.e., the same interface), but with different features and implementations. Object-oriented programming techniques provide a means to solve this requirement with the least effort.
6.2 Design Objectives

If a user gains access to a new cluster and wants to use it to migrate applications to it, the migration framework should not require administrative rights for installing software or kernel modules on the new system. We achieve this by implementing OGRE in user-space only. OGRE does neither require any kernel modules to be installed on a cluster nor that software is deployed to the cluster. Additionally, OGRE does not depend on any libraries on the target systems. OGRE is self-contained, that is, a single binary is sufficient to activate a cluster for application migration.

The applications that are subject to migration are not known upfront to the migration framework. Moreover, OGRE must not be restricted to only migrate applications that are compiled with the reparallelization and checkpointing system presented in this work. To be useful to Grid research, OGRE needs to handle any application written in any programming language or programming model. Hence, the interface of OGRE has to be slim, that is, using the framework should not result in compiling and linking the complete migration framework into the application. In addition, the interfaces to OGRE should not prescribe a particular programming language for the implementation of the application.

As the collectivity of the Grid typically exhibits a heterogeneous computing platform, OGRE is required to be platform-independent as well. OGRE should run on any platform that is accessible throughout the computational Grid. Therefore, we have chosen to implement OGRE in Java, as a JVM is likely to exist on all architectures that may be used in a computational Grid. We can even assume that a JVM is already preinstalled on the accessible systems. As Java follows the “compile once, run everywhere” approach, it is unnecessary to recompile OGRE if a new system becomes available to the user. With Java, it is sufficient to transfer the OGRE executable to the new system and launch it there.

Related to the issue of flexibility and platform independence, OGRE must support different cluster management systems, such as OpenPBS [14] and the Sun Grid Engine [189]. We solve this problem by introducing an OGRE driver component for the cluster management systems. The driver component abstracts from the concrete interface to the cluster management system and provides a generic interface to OGRE’s internal components.

Grids typically form a loosely coupled system, that is, each cluster of the Grid is a separate entity that is managed by the organization owning the cluster. Hence, clusters may join and leave the Grid at any time (e.g. by
rebooting the system or blocking accesses). The loose coupling of systems has to be respected by the OGRE framework as well. Client/server solutions are inappropriate, since they require a distinguished system (the server) that is constantly accessible by the clients throughout the whole Grid infrastructure. Obviously such a server system is subject to the loose coupling of the infrastructure as well and it cannot be guaranteed that the server is constantly accessible.

Peer-to-Peer (P2P) networking [15] provides a solution, in which no central servers exist and each client plays the role of a server. As there is no distinguished server system, all participants of the P2P network can arbitrarily join and leave the P2P network; this directly corresponds to the nature of the Grid infrastructure. OGRE follows the P2P approach and considers each OGRE instance as a P2P client with the same functions as the other. Only the OGRE instance that currently monitors a running application is treated in a special way; the monitor instance is used to determine the migration plan for the running application (see Section 6.4).

6.3 The OGRE Migration Framework

In this section, we present the architecture and the high-level functions of the OGRE migration framework. The architecture is designed such that it fulfills all the requirements set up in the previous section. To achieve the required flexibility, each component is implemented as a plugin. The plugins can be changed to a different implementation by replacing the corresponding factory [74] of the component.

6.3.1 Components

Figure 6.1 depicts the architecture of the migration framework. Each block corresponds to a component that implements a particular internal function of the OGRE framework. The arrows indicate message flows between different parts of the framework. A thick arrow denotes the direction of messages that are sent through network connections. Thin arrows identify the direction of in-process communication, that is, method invocations or signaling (i.e., Object.wait() and Object.notify()) between different threads of the framework.
6.3 The OGRE Migration Framework

The application adapter is used for the intra-cluster communication between OGRE and migratable applications. The application adapter provides the required thin layer that has to be included in the application. The remainder of the migration framework runs in a separate daemon process on one of the front-end nodes of the cluster or any other machine that is directly accessible throughout the cluster. In a Java application, the application adapter uses regular Java sockets [88] to connect to its server part (the application communicator) that interfaces with the internals of OGRE. In other programming languages, the program can use system calls of the operating system and open a network connection to the application communicator’s interface.

For message transport, standard Java object serialization [184] cannot be used to serialize and deserialize the messages, as this would limit OGRE’s abilities to support non-Java applications. Instead, the adapter and communicator use a slim communication interface that provides limited serialization/deserialization features (e.g. support for tree-shaped data structures only) to a Java application, but that can be easily ported to applications written in other programming languages (e.g. C/C++ or Fortran). The limited serialization/deserialization approach is sufficient, as the messages are simple and only contain a small amount of data (e.g. file names, time stamps, etc.). Messages are sent asynchronously between the application...
adapter and the communicator, that is, the receiver does not respond to a message with an acknowledgment.

Inter-cluster communication is performed by the \textit{P2P communicator}. Each OGRE instance is connected to each other through a P2P-style interconnection network on top of the WAN that connects the different cluster sites. Each instance of our prototype uses the SmartSockets library [132] for connecting to the P2P network and communicates with each other by asynchronously sending typed messages. Java object serialization is applied to convert the message objects into a byte stream that is sent through the P2P network connections. The current prototype implementation uses the SmartSockets [132] communication library that is designed to interconnect clusters sites that are hidden behind firewalls or are located in private networks.

The migration framework has to search for free resources on each of the accessible systems and needs to submit jobs to the clusters’ schedulers. This is accomplished by the \textit{batch system communicator}, which acts as a driver component to hide the interfaces to the various cluster schedulers behind a common interface. The batch system communicator reads the clusters schedule (e.g. by parsing log files or executing a command for status queries) and reports the reserved PEs to the \textit{filter and scheduler} component. In turn, the batch system communicator receives submission requests from the \textit{migrator} and accesses the cluster’s job scheduler to actually launch the application.

The filter and scheduler receives the current schedule of the cluster and analyzes it for free PEs (so-called \textit{reservation holes}). After the scheduler has created the list of reservation holes, it passes the list to its filter sub-component. The filter performs postprocessing on the reservation holes and removes reservation holes that are not suited for receiving a migrated application. For example, a reservation hole is unusable if its runtime limit is too small to effectively start an application in it. After postprocessing, the list of reservation holes is sent through the P2P network to the central OGRE instance, that is, the OGRE instance that currently monitors a running application. The message contains the number of PEs of each reservation hole, the starting time of the hole, and how long the reservation hole lasts.

The central OGRE instance receives the lists of reservation holes (the so-called \textit{bids}) of all other OGRE instances and passes it to the \textit{auctioneer}. The auctioneer computes the application’s \textit{migration plan} that describes when and where the application is migrated to. Before the auctioneer determines the migration plan for the application, the list of reservations is again filtered.
The auctioneer uses a set of constraints that can be defined by the application and discards reservations that do not fulfill these constraints. An example constraint is the minimum number of PEs the application requires to run.

After discarding reservation holes violating the application constraints, the auctioneer instantiates an auction algorithm (see Section 6.4) and passes the remaining reservation holes to it. The auction algorithm evaluates the reservation holes and selects the subset of the reservation holes that constitute the best migration plan. The migration plan is then forwarded to the migrator for execution.

The migrator broadcasts the computed migration plan to the other OGRE instances. The migrators on the remote framework instances accept the migration plan and, depending on the migrator heuristics, submit jobs to their local cluster. Submissions are only issued if the migration plan includes reservations for a migrator’s local system. To submit the job, the migrator contacts the batch system communicator and requests the given number of PEs for the time specified by the migration plan entry.

### 6.3.2 Application Life Cycle

The migration framework fully controls the application’s life cycle. When the application starts, the framework takes over control and releases control when the application successfully terminates computation. This is necessary as OGRE decides when the application needs to be is checkpointed and on which cluster it is resumed afterwards. During its life time, the application evolves to four different states (see Figure 6.2).

An application’s execution in the OGRE framework starts when a user arbitrarily chooses an OGRE instance and submits the application to it. After submission the application is in the state submitted and waits for execution on one of the accessible clusters. The OGRE daemon that received the application’s submission request becomes the central node and starts to monitor the application. The OGRE daemon then requests bids from all clusters and creates a first migration plan for the application (see Section 6.4). According to the computed migration plan, the application is sent to the cluster that is first in the migration plan. The application stays in the submitted state until the first reservation for the application on of the cluster starts.
Figure 6.2: States of the application life cycle in OGRE.

When the first reservation for an application becomes available on one of the accessible clusters and the application starts executing, the application's state is changed from submitted to running. The new state indicates that the application is actively performing computations on one of the clusters and needs to be checkpointed if a migration is scheduled later on. An application in the submitted state does not require OGRE to force creation of a checkpoint, as the application is waiting for start-up and thus does not yet have a computational state associated with it.

At any time the application needs to be moved to another system, the central OGRE daemon informs the checkpointing runtime of the application about the migration and a checkpoint is created. After successfully creating the checkpoint, the checkpointing runtime aborts the application and informs the OGRE daemon about the storage location of the checkpoint data on disk. The OGRE daemon assigns the checkpointed state to the application, indicating that the application is not actively computing, but sleeping in a checkpoint for migration. When the application resumes operation, it is reassigned the running state.

The cycle of state transitions of the applications’ state between running and checkpointed continues until an application has successfully finished execution and regularly terminates its computation. OGRE notices this event and sends the application to the finished state to indicate that further migrations of the application are no longer necessary. OGRE does not consider finished applications for the creation of migration plans and only assigns free reservation holes to other applications that are not yet finished.
6.4 Migration Strategy

As a proof of concept, we have implemented a simple migration strategy. The key assumption of our strategy is that the user of an application is interested in receiving the final results of the application as fast as possible.

On the one hand, the computing power delivered to an application should be maximized to speed up the application as much as possible. This can be achieved by migrating the application to the cluster that offers computing power at the earliest time. If the application has not finished yet, the next cluster is chosen in the same way. If two clusters offer computing power at roughly the same time, the more powerful cluster receives the application.

On the other hand, the number of migrations should be minimized, as each migration introduces a cost for checkpointing, for copying the checkpoint data to the target, and for resuming the application from the checkpoint. If a reparallelization is needed, additional costs incur, as after reparallelization data locality is lost and the Jackal DSM has to re-establish the optimal data distribution before the application can continue at full speed.

It is the task of our migration strategy to find a tradeoff between the number of migrations and the computing power consumed by the application. In the following, we first present how our strategy assesses the performance of the individual clusters (Section 6.4.1). We then describe how to quantify the computing power of a single cluster reservation (Section 6.4.2). A benefit function for assessing the benefit of migration is introduced in Section 6.4.3. Finally, we demonstrate a graph-based algorithm to compute a migration plan (Section 6.4.4).

6.4.1 Quantifying the Performance of Clusters

To determine the computing power a cluster may deliver to a given application, it is necessary to quantify the performance an application can achieve on a particular cluster. Unfortunately, the computing power is not determined by the cluster’s key data, such as clock frequency, main memory size, or disk bandwidth. Performance strongly depends on algorithms that the application uses for computing its results. A memory-bound application, that is, an application that transfers a large amount of data to and from the main memory, will not exploit a CPUs peak performance. Vice versa, a
6 Framework for Inter-cluster Migration

Figure 6.3: Cuboid quantifying the computing power of a cluster reservation.

CPU-bound application only needs a small working set of data to run and, thus, does not saturate the memory sub-system.

Estimating the application’s runtime behavior is a very complex task and is subject to current research [10, 11, 64, 167]. We thus restrict ourselves to a very simplistic performance model for applications. Due to the component nature of OGRE, other performance models can easily be added to the migration framework later if available and if the simple performance model is not sufficient. Currently, OGRE relies on the user to quantify the performance of the cluster, since fully automatic solutions do not yet exist.

For each cluster OGRE expects a single file containing a so-called cluster profile. The cluster profile contains the total number of nodes in the cluster, the number of processors per node, and a number that indicates the speed of the CPUs. We assume that this single source of information is sufficient per cluster, since most clusters are homogeneous systems and do not contain a mixture of different node and CPU types.

In addition to the performance data, the profile also contains information about the maximum permissible walltime limits for jobs submitted to the cluster. The user can specify a day time limit and a night time limit, if the cluster enforces different wall times for jobs during day or night time.

6.4.2 Quantifying the Performance of Cluster Reservations

Using the presented simplification to the performance model, we can now investigate how OGRE models the performance of a cluster reservation.

We can express the computing power of a cluster reservation $r$ as a 3D cuboid (see Figure 6.3). The length of the cuboid is determined by the
Walltime limit $t_r$ of the reservation. The depth corresponds to the number of processing elements ($p_r$); the speed $s_r$ of a PE is used as the cuboid’s height. Additional performance measures can be added as new dimensions, effectively extending the cuboid to form a $d$-dimensional hyper-cuboid, where $d$ is the total number of performance measures.

Assuming that the application’s speed scales perfectly with the number of processing elements, the computational power $P(r)$ of a reservation $r$ is then determined as the volume of the cuboid:

$$P(r) = s_r \cdot p_r \cdot t_r$$  \hspace{1cm} (6.1)

If a speed-up profile of the application is available for the cluster, the number of PEs $p_r$ can be replaced by the speed-up $S_{p_r}(N)$ expected for a given PE count. This yields a better estimation of the cuboid’s volume. A definition of the term speed-up is given in Chapter 7 or in [102].

This quantification scheme is applied to assess the computing power a reservation hole is expected to deliver to the application. For each reservation hole, the filter and scheduler component reports the size and shape of the cuboid to the central OGRE instance, which computes the volume of the cuboid and passes the reservation hole and the volume to the auctioneer.

### 6.4.3 Benefit Function

The quantification scheme for reservations is used to define a benefit function that determines the benefit an application gains if it is migrated to a particular reservation. While executing the application on a cluster reservation yields a benefit, migrations cause a runtime penalty that can be expressed by costs that in turn reduce the benefit.

Hence, if the application migrates from one reservation $r_1$ to another reservation $r_2$ on the same cluster (intra-cluster migration), we can express the benefit as:

$$B_{\text{local}}(r_1, r_2) = P(r_2) - (C_{\text{cp}}(r_1) + C_{\text{rs}}(r_2) + C_{\text{re}}(r_2))$$  \hspace{1cm} (6.2)

where $C_{\text{cp}}(r)$ is the costs of checkpointing with respect to a reservation $r$, $C_{\text{rs}}(r)$ expresses the costs of resuming the application from the checkpoint, and $C_{\text{re}}(r)$ summarizes the costs induced by reparallelization. Transferring the checkpoint to a remote system is not necessary.
As the application blocks and does not make any progress during checkpointing, \( p_r \) PEs of speed \( s_r \) do not execute the application for the time \( t_{r,\text{cp}} \) that is needed to create the checkpoint (a similar argument holds for \( C_{rs} \)). Thus, the cost function for checkpointing and resuming the application evaluate to:

\[
C_{\text{cp}}(r) = s_r \cdot p_r \cdot t_{r,\text{cp}} \tag{6.3}
\]

\[
C_{\text{rs}}(r) = s_r \cdot p_r \cdot t_{r,\text{rs}} \tag{6.4}
\]

Similarly, the costs for reparallelization can be estimated as:

\[
C_{\text{re}}(r) = s_r \cdot p_r \cdot t_{r,\text{re}} \tag{6.5}
\]

where \( t_{r,\text{re}} \) is the time the application blocks during reparallelization plus the time passing by until the DSM system has re-established the data distribution. \( C_{\text{re}} \) constitutes an upper bound, as the application makes progress during data relocation and is not blocked.

For an inter-cluster migration, that is, a migration between reservations on different clusters, an additional cost reduces the benefit. However, the migration costs need to be considered, if the target reservation and the source reservation are active while the checkpoint is still transferred to the target system. Again, we can express the costs as a loss of computing power:

\[
C_{\text{mg}}(r) = s_r \cdot n_r \cdot t_{r,\text{mg}} \tag{6.6}
\]

where \( t_{r,\text{mg}} \) is the time needed to complete the checkpoint transfer while the target reservation is already active. Hence, the benefit of a remote migration between two reservations \( r_1 \) and \( r_2 \) on different clusters can be expressed as:

\[
B_{\text{remote}}(r_1, r_2) = B_{\text{local}}(r_1, r_2) - C_{\text{mg}}(r_2) \tag{6.7}
\]

### 6.4.4 Computing the Migration Plan

With the aforementioned quantification scheme, we are ready to describe the default strategy for application migration in our prototype implementation of OGRE.

From all incoming reservation holes, the auctioneer selects the ones that deliver the highest amount of computing power (according to the quantification scheme) and cause a minimal number of migrations. The problem of finding
6.4 Migration Strategy

Figure 6.4: Example of incoming bids containing reservation holes.

an optimal migration plan can be reduced to finding the longest path in a graph. In the following, we will present how the incoming bids are mapped to a Directed Acyclic Graph (DAG).

Computing the migration plan is performed in two steps: (1) constructing the so-called migration graph and (2) finding a migration path in the migration graph.

Construction of the Migration Graph

As mentioned before, the auctioneer first discards all reservation holes that do not fulfill the application’s constraints. It then creates a graph structure from the reservation holes. Figure 6.4 depicts an example of a set of reservation holes that are reported by two clusters. Each reservation is depicted as cuboid (depicted as a rectangle for clarity) and arranged by the corresponding starting time of the reservation hole. Each cluster A and B reports three reservation holes of different shape. In case of Figure 6.4, the optimal migration strategy is to start the application at B.1 (on cluster B) and then proceed to cluster A to use reservation A.3.

From the set of bids, we can construct a migration graph \( G \). For each reservation \( r \) a vertex is created in the graph. If it is possible to migrate
Figure 6.5: Graph representation of the bids in Figure 6.4.

an application from reservation $r_1$ to $r_2$, the vertices corresponding to $r_1$ and $r_2$ are connected by a weighted edge (see below). Figure 6.5 shows the graph that results from the bids of Figure 6.4. The added migration edges are depicted as solid arrows, the vertices as circles. More formally, an edge $(v, w)$ between two vertices $v$ and $w$ is added if and only if the reservation $v$ ends before $w$ starts.

Our algorithm adds two additional pseudo vertices $S$ and $T$ to the graph. $S$ is called a root vertex, that is, a vertex without incoming edges and $T$ is called a leaf vertex (without outgoing edges). The pseudo vertices make the handling of the graph easier as $S$ is the only root vertex and the vertex $T$ is the only leaf vertex. With these vertices added to the graph, a migration path is defined as the longest path $S \rightarrow^* T$ from $S$ to $T$.

For each inner vertex $v$, a pseudo edge $(S, v)$ that connects $v$ and $S$ is added. Each inner vertex $w$ is connected to $T$ by an edge $(w, T)$. Figure 6.5 indicates the added pseudo edges by dashed gray arrows. The edge $(S, v)$ to an inner vertex $v$ expresses that a submitted or checkpointed application can be started in any reservation hole in the migration graph. Hence, $S$ is connected to any other vertex. If the application is scheduled to run on a reservation $w$, it may be sent to the checkpointed state or terminate. Hence, there is an edge $(w, T)$ connecting $w$ and $T$.

The benefit function of Section 6.4.3 is evaluated once for each edge of the graph and the result is assigned as the edge’s weight. If two vertices $v$ and $w$ correspond to two reservations on the same system, $B_{\text{local}}(v, w)$ deter-
6.4 Migration Strategy

mines the weight of the edge \((v, w)\). If two vertices stem from reservations on different systems, \(B_{\text{remote}}(v, w)\) is evaluated. The pseudo edges have a weight of zero by default, as migrations along these edges do not provide computational benefit to the application.

Computing the Migration Path

The migration graph contains all information needed to compute the migration plan from it. To recapitulate, the vertices reflect reservation holes that can receive the application and the edges express possible migrations between these reservation holes. The benefit of a migration is stored as the weight associated with the edges.

Thus, the migration path \(S \rightarrow^* T\) that corresponds to the highest benefit for the application can be found by searching for the longest path in the migration graph. In our case, the longest path is defined as a path in the migration graph that obtains the highest benefit (in terms of the benefits associated with each edge in the graph). The graph is a DAG, as it is guaranteed that there is no path starting in \(v\) that reaches \(v\). If such a path existed, it would correspond to the information that the application could travel backwards in time. In a DAG, the longest path can be computed by sorting the graph topologically [77] and running the longest path algorithm to retrieve the longest path \(S \rightarrow^* T\). Both algorithms have a complexity of \(O(|V| + |E|)\), where \(|V|\) is the number of vertices in the graph and \(|E|\) is the number of edges.

Figure 6.6 depicts the longest path algorithm as Java-like pseudo-code. The algorithm expects the graph, the root vertex \(S\), and the leaf vertex \(T\) as input parameters. Line 4 creates a hash map that is later used to store the partially constructed longest path. As the algorithm constructs the path in a piecewise fashion by continuously updating the end of the path, the hash map effectively stores the longest path in reverse order. In line 6, the algorithm creates the topological ordering of the graph’s vertices. It then initializes an array that stores partial results of the benefits associated with the partially constructed longest path.

Lines 8–17 iterate over the vertices \(v\) in topological order and construct the longest path. For each outgoing edge \((v, w)\) of \(v\), the algorithm compares the current benefit of a path that reaches \(w\) and the new benefit of a path
6 Framework for Inter-cluster Migration

```java
class LongestPathAlgorithm {
    public static Path longestPath(Graph graph, Vertex S, Vertex T) {
        HashMap<Vertex, Vertex> parent = new HashMap<Vertex, Vertex>();
        Vertex[] order = topologicalSort(graph, S);
        double[] benefit = new double[levels.length];
        for (Vertex v : order) {
            for (Edge e : graph.getOutgoingEdges(v)) {
                Vertex w = e.getTarget();
                double b = e.getBenefit();
                if (benefit[v] + b > benefit[w]) {
                    benefit[w] = benefit[v] + b;
                    parent.put(w, v);
                }
            }
        }
        return Path.reverse(parent, T);
    }
}
```

Figure 6.6: Longest path algorithm.

through v. If the benefit of the new path is higher than the currently highest benefit of a path reaching w, the algorithm updates the benefit and adds the reversed edge (w, v) to the hash map. During the course of the algorithm, the hash map is filled with reversed (partial) longest paths found so far.

After the algorithm has finished processing the last vertex of the graph (i.e., the leaf vertex T), line 18 starts with the leaf vertex T and traces the hash map to find the longest path to the root vertex S by reversing the edges that have been added in line 14. The auctioneer then converts the migration path found into the migration plan. It discards the pseudo vertices added during construction and maps the vertices in the path back to their corresponding reservation holes.

After the auctioneer is finished with computing the migration plan, it broadcasts the migration plan to all migrators in other instances of the framework. This ensures that each migrator knows about all reservations an application might take during its life time. In the following section, we are going to discuss how a migrator uses the migration plan to eventually migrate the application.
6.5 Application Migration

Each migrator on a cluster traces the migration plan and retrieves all reservation holes that affect the local cluster. If a reservation for the local cluster is present, the migrator contacts the batch system communicator (see Figure 6.1) and issues a job submission to it. The batch system communicator creates an adequate job script for the cluster management and submits the script to the queueing system. Figure 6.7 shows an example job script as it is submitted by the batch system communicator. The job script launches a restricted OGRE daemon (the monitor) that is responsible for starting the application and monitoring its execution.

In line 3 of Figure 6.7, the job contains a resource request for the reservation. In the example, the script requests four nodes for seven hours and 35 minutes. The script then starts the OGRE monitor in line 6. The first argument is an ID that corresponds to an entry in the migration plan and uniquely identifies all reservations that are made at a cluster. The second argument (“–walltime”) informs the monitor about the maximum permissible runtime of this reservation. Then, the script passes arguments that help the monitor contact the OGRE daemon. The monitor expects the name of the machine that executes the OGRE daemon (“–cluster-ogre-daemon”), the name of the cluster (“–cluster”), and the application’s name (“–appname”). Finally, the command line to start the application is passed as string.

Figure 6.7: Example of a job script for submission.

```
#!/bin/bash

PBS -l nodes=4,walltime=07:35:00

export OGRE_ROOT=/home/user/ogre
/home/user/ogre/bin/monitor ogre-migration -l
    --walltime 07:35:00
    --cluster-ogre-daemon frontend.clusterA
    --cluster clusterA
    --appname anApplication
    --command "application --dowork"
```
With the information on the command line, the monitor contacts its local OGRE instance and sends a message, informing it about the activated reservation and that it is ready to receive the application. Three cases may occur:

1. The application is still in submitted state. The central OGRE instance then replies with a message that causes the monitor to start the application.

2. The application is not running on a different cluster and is thus in the checkpointed state. The central OGRE daemon migrates the application to the activated reservation by copying the checkpoint data to the corresponding PEs. The monitor is then informed to resume the application from the checkpoint.

3. The application is still running on a remote cluster. Since we consider reservations as an atomic execution unit, OGRE does not prematurely abort the application on the remote cluster. Instead, the monitor is informed to release the reservation and to terminate.

In the third case, the terminated reservation is discarded from the migration plan. If the migration plan is empty after removing the terminated reservations, a new migration plan is computed from scratch, effectively updating it to the current state of the individual clusters.

To restart the application from a checkpoint, the OGRE daemon of the local cluster copies the individual checkpoint files to their respective machines and launches the application. The checkpointing runtime automatically detects the existing checkpoint files and initiates the restore algorithm that reads the checkpoints and reconstructs the application’s state. After the restoration is completed, the application continues execution as usual. During execution, the checkpointing runtime waits for an incoming migration signal.

6.6 Summary

In this section, we have presented the prototype of a framework for automatic application migration. We have identified requirements for the framework and argued about possible implementation approaches.

We have explicitly designed the framework for a maximum of flexibility to enable further research on migratable applications. The implementation
should be platform-independent and self-contained to avoid recompilation and installation of the migration framework on the target systems. OGRE supports any programming language by providing its functions through a thin network interface that can be accessed from any programming language. To cope with the loose nature of Grid resources, OGRE relies on a P2P-shaped network to interconnect OGRE instances running on different Grid resources.

For the prototype, we have proposed a component-oriented architecture in which each of the components can be easily exchanged by other components. This enables researchers to freely adapt the control flow of the OGRE framework to their needs.

To test the migration capabilities of the framework, we have implemented a default migration strategy. The strategy greedily requests resources and tries to maximize the computing power delivered to the application while minimizing the number of migrations. The incoming reservation holes are assessed for their migration benefit by applying a simplistic quantification scheme for cluster performance. From the reservation holes, OGRE constructs a migration graph and uses the benefit function to weight the edges of the migration path. A longest path algorithm then computes the migration path with the highest benefit of the individual migrations.
We evaluate the performance of the JaMP implementation with a set of micro-benchmarks, a subset of the Java Grande Forum benchmarks, and a compute-intensive fluid simulation. The environment used for benchmarking is a commodity cluster (named Woodcrest) of quad-core (2×2 cores) Intel Woodcrest machines (clock frequency 3.0 GHz) and 8 GB of main memory per machine. The nodes are connected by Gigabit Ethernet and InfiniBand 4x in full-duplex mode. The nodes run Debian Linux (kernel version 2.6.5-7.283-smp). The cluster is maintained by the computing center of the Friedrich-Alexander-University Erlangen-Nuremberg, Germany.

For each benchmark, we compute the runtime as the average of up to five independent runs. For the measurements, we only use one CPU of each machine and configure Jackal to use the InfiniBand interconnect. Hence, network effects are visible in the runtimes and provide a better insight into the speed-ups that can be achieved with our JaMP implementation.

The baseline for the speed-up measurements is the runtime of the benchmarks with only one CPU. We define the speed-up \( S_P(N) \) of a particular benchmark application as \[136]\):

\[
S_P(N) = \frac{T_1(N)}{T_P(N)} \tag{7.1}
\]

In the above formula, \( T_P(N) \) is the time needed to execute the benchmark with data size \( N \) on \( P \) processing elements.

In literature, the speed-up is often defined in a more restricted fashion by using \( T^*(N) \) instead \( T_1(N) \) as the baseline \[102\]. The time \( T^*(N) \) is the runtime of the fastest known sequential algorithm applicable to solving the
problem of size $N$. We use the weaker speed-up definition, as we are interested in the speed-up gain by our JaMP implementation and are not considering speed-ups achieved by different parallel algorithms.

In Section 7.1, we evaluate the overheads of the basic JaMP operations such as the creation of parallel regions. We then introduce the set of benchmarks used for our experiments in Section 7.2. Section 7.3 shows the performance that can be achieved for the benchmark applications. The effects of reparallelization on the benchmarks are discussed in Section 7.4. Section 7.5 demonstrates the overheads that the checkpointing package introduces. Finally, we assess the results that where achieved by migrating a compute-intensive application between two clusters (Section 7.6).

### 7.1 Microbenchmarks

To determine the speed of the basic JaMP operations, we use the same set of microbenchmarks that has been used to assess the JOMP implementation [38]. The benchmarks measure the runtime costs of creating parallel regions and of executing synchronization directives. Additionally, the microbenchmarks evaluate the costs of different scheduling types with different chunk sizes. As suggested in [34], the microbenchmarks compute the overhead of a particular directive by measuring the runtime of the execution of an empty loop without any OpenMP/Java instrumentation and the runtime of the same loop with one of the OpenMP/Java directives added. The runtime costs of a particular OpenMP/Java construct then evaluates to the difference of both runtimes.

To discuss JaMP’s runtime costs that are caused by the OpenMP/Java statements, we first determine the average latency of a single message on the network. Jackal’s communication is mostly performed in a request/reply fashion. Thus, we consider the ping-pong latency, that is, the time from sending a request message until the corresponding reply message is received. The ping-pong latency of a Jackal-internal RPC on the Woodcrest cluster is roughly 57 $\mu$sec on InfiniBand connections. The average message latency thus is roughly 28.5 $\mu$sec.

To determine the overhead of accessing shared JaMP variables compared to regular instance variables, we touched both types of variables $10^6$ times in separate loops. We then subtracted the loop overhead from both results.
An access to a shared variable is about 1.8 times slower than an access to an instance variable. The higher latency of the shared variable access is caused by the fact that it resides in an array in the \texttt{JampParam} object and therefore requires two indirect memory accesses instead of one. For shared variables, the values may not be cached locally as many threads may access the variable concurrently. In contrast, \texttt{firstprivate} variables can be cached after their initialization; subsequent accesses are overhead-free.

The runtime cost to start and finish an empty parallel region (\texttt{parallel} construct) is about 24 msec on eight nodes (see Figure 7.1(a), triangle line) and increases to about 50 msec on sixteen nodes. The runtime costs consist of (1) creating and initializing the shared and private instances of \texttt{JampParam} to transport the values of variables, (2) instantiation of the thread team, and (3) the implicit barrier synchronizations (roughly 2.1 msec for eight nodes and 4.4 msec for sixteen nodes, see below) at the end of the region. The thread team is created sequentially, that is, a loop creates one thread at a time. Internally, a sequence of Jackal RPCs is triggered that create and start the threads on the remote nodes. Additionally, a large amount of DSM protocol activity is caused by the remote threads during start-up. The DSM protocol allocates the read/write bitmaps for the new thread and initializes the bitmaps.

Two barriers mainly contribute to the runtime costs of the \texttt{for} construct (diamond line). They contribute roughly 9 msec to the total runtime costs on sixteen nodes. The first barrier waits for the initialization of the for loop’s \texttt{LoopInformation} object; the second barrier is required by the OpenMP specification. The DSM protocol is responsible for the remaining 1 msec, as it initializes and distributes a copy of the \texttt{LoopInformation} object on sixteen nodes. The runtime costs of the compound \texttt{parallel for} (square line) consist of the time needed to execute the \texttt{parallel} and \texttt{for} region.

The overhead of the \texttt{barrier} statement (see Figure 7.1(b), triangle line) is due to Jackal’s naive linear barrier implementation. Whenever a thread reaches the barrier, a \texttt{synchronized} method is invoked and a counter is incremented. The thread then issues an \texttt{Object.wait()} and blocks until the barrier’s goal is reached. This implementation of the barrier causes a high DSM protocol overhead, as Jackal retrieves the barrier object from its home node and flushes it for every thread entering the barrier. For sixteen threads, this causes 64 messages on the network, which contribute 4 msec of 4.4 msec for a barrier on sixteen nodes. Since for large numbers of nodes this
7.1 Microbenchmarks

(a) Runtime costs of OpenMP/Java constructs.

(b) Runtime costs of OpenMP/Java directives.

Figure 7.1: Runtime costs of OpenMP/Java constructs and directives.
barrier algorithm will eventually become a serious bottleneck, a hierarchical implementation with better scalability should be added to Jackal’s DSM in the future.

For the reduction clause (Fig. 7.1(b), square line), the runtime costs mainly consist of the time needed to combine the partial results of the workers. For eight machines, the time needed for a +-reduction of a variable of type `long` is roughly 1.7 msec. For sixteen machines, the costs increase to 4.6 msec due to the sequential reduction algorithm that is implemented in the JaMP runtime.

Figure 7.2 displays the overheads of the different scheduling sizes of parallel for loops with static and dynamic scheduling. We measure 8, 64, and 512 loop iterations per chunk. For a given number of nodes, there is virtually no difference in the runtime costs for different chunk sizes in static loops (see Figure 7.2(a)). The threads receive a read-only copy of the `LoopInformation` object and can compute the static loop boundaries from the data contained in the `LoopInformation` object.

In contrast, the runtime costs of a dynamic loop strongly depend on the chunk size (see Figure 7.2(b)). For such loops, a thread conceptually retrieves the next block from a list of unprocessed chunks. Although the list can internally be implemented by counters, each access to the `LoopInformation` object has to be synchronized to avoid race conditions between concurrent accesses of different threads. This causes a significant amount of DSM protocol activity to maintain consistency of the loop information across the threads. The runtime cost of the central loop information decreases with growing chunk sizes, since less and less blocks need to be requested.

This evaluation of the microbenchmarks for the thread binding shows that our implementation of JaMP in the Jackal DSM is efficient. All runtime costs attribute to the DSM protocol and are completely unrelated to the implementation of the JaMP runtime system.

### 7.2 Benchmarks

For the evaluation of the JaMP implementation, the reparallelization runtime, and the checkpointing runtime, we use two kinds of benchmarks. First,
7.2 Benchmarks

Figure 7.2: Runtime costs of scheduling types in OpenMP/Java.

(a) Runtime costs of static scheduling.

(b) Runtime costs of dynamic scheduling.
Table 7.1: Data sizes used for the benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size A</th>
<th>Size B</th>
<th>Size C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypt (JGF)</td>
<td>12,000,000</td>
<td>80,000,000</td>
<td>200,000,000</td>
</tr>
<tr>
<td>Series (JGF)</td>
<td>40,000</td>
<td>400,000</td>
<td>4,000,000</td>
</tr>
<tr>
<td>SOR (JGF)</td>
<td>4,000</td>
<td>6,000</td>
<td>8,000</td>
</tr>
<tr>
<td>Sparse (JGF)</td>
<td>200,000</td>
<td>400,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Raytracer (JGF)</td>
<td>600</td>
<td>2,000</td>
<td>n/a</td>
</tr>
<tr>
<td>LBM</td>
<td>100</td>
<td>200</td>
<td>n/a</td>
</tr>
</tbody>
</table>

we use the Java Grande Forum (JGF) benchmark suite [37, 175] that is parallelized for the JOMP OpenMP/Java compiler. We modified the benchmarks and replaced the JOMP pragmas with our OpenMP/Java syntax. Second, we evaluate the compute-intensive Lattice-Boltzmann Method (LBM) [210] as benchmark.

This section provides an in-depth introduction of the benchmark codes and describes the parallelization technique applied to the codes. From the JGF benchmarks we study section 2 (numeric kernels) and section 3 (applications). We skip section 1 of the benchmark suite, since it solely contains micro-benchmarks for individual OpenMP directives. From sections 2 and 3, we study Crypt, Series, SOR, Sparse, and Raytracer. We skip LU Fact and Monte Carlo, since they are not suitable to be executed on a DSM on a cluster [116]. Euler has to be left out, as it uses the currently unsupported OpenMP construct ordered.

Table 7.1 shows the size of the data sets that are processed by each benchmark. For each benchmark, we demonstrate the effect of three different data sizes A–C. For the compute-intensive Raytracer benchmark, the JGF suite does not define a large data size C. Following the JGF benchmarks, we use only two input sizes for the LBM benchmark as well.

Crypt (section 2) performs IDEA encryption and decryption of $N$ bytes stored in an array of byte values. Crypt contains two tight loops, one for encryption and one for decryption of the input data. Each loop strongly depends on bit and byte operations. The average runtime per loop iteration is very low compared to the overall runtime of the benchmark. The encryption
loop and decryption loop are parallelized using the `parallel for` construct without a `schedule` clause.

**Series** (section 2) computes the first \( N \) Fourier coefficients of the function \( f(x) = (x + 1)^x \) on the interval \([0..2]\). The benchmark mainly uses transcendental and trigonometric functions to compute the coefficients, which makes Series a highly compute-intensive benchmark. The main loop of Series is parallelized with the `parallel for` construct and static scheduling with default chunk size.

**SOR** (section 2) is a simple over-relaxation in a red-black style on a 2D grid of \( N \times N \) cells of data type `double`. SOR performs 200 relaxation steps on the grid and is highly memory bound. The inner loop is parallelized in column order by means of a `for` construct. The outer loop along the grid rows is parallelized with the `parallel` construct to avoid excessive thread creation and termination for each SOR step. After each SOR step, a `barrier` synchronization is required to avoid race conditions. We also have parallelized the data allocation using a `parallel for` construct such that the nodes that work on a partition of the grid also perform the allocation. This is a well-known optimization technique for OpenMP programs on NUMA systems and is also applicable for COMA-like DSM systems such as Jackal or TreadMarks.

**Sparse** (section 2) computes the product of two sparse \( N \times N \) matrices in compressed-row format [209]. Sparse repeats the matrix multiplication 200 times to achieve enough runtime for reliable measurements. Sparse is a memory-intensive benchmark that exhibits a high degree of indirect and non-regular memory references that stem from the compressed-row format. The benchmark loop is parallelized with the `parallel` directive; work is assigned by means of the OpenMP/Java runtime functions `Omp.ompGetNumThreads()` and `Omp.ompGetThreadNum()`.

**Raytracer** (section 3) renders a picture that contains a scene of 64 spheres. The resolution for a benchmark run is \( N \times N \) pixels. The picture is flattened into a 1D Java array that is shared among the worker threads. Raytracer is a pleasingly parallel application (sometimes also called “embarrassingly parallel application”), that is, during computation virtually no communication between the threads takes place. Only at the end of the benchmark run, the picture array is written back to the home node of the array. The main loop of the benchmark is parallelized with the `parallel for` directive.
with dynamic scheduling (chunk size 10), effectively assigning stripes of the picture to each of the threads.

**LBM** uses cellular automata to numerically simulate nearly incompressible fluids with speeds of low Mach numbers. Space and time are discretized and normalized. The benchmark computes 50 time steps over the 3D grid and operates on a 3D domain divided into $N \times N \times N$ cells that hold a finite number of states (D3Q19 model). In one time step the complete set of states is updated synchronously by deterministic, uniform update rules. For parallelization, we use the `parallel` construct for the time-stepping loop. The loop over the $x$-axis of the 3D grid is augmented with the `for` directive (default scheduling). Similar to SOR, we also parallelized the data allocation using `parallel for`.

### 7.3 OpenMP/Java

In this section, we evaluate the performance of the JaMP implementation with the set of benchmarks of Section 7.2. For each benchmark, we measure the runtime of the benchmark kernel on one to sixteen nodes with one thread per node. We first show the benchmark results for the thread binding of JaMP in Section 7.3.1. We then demonstrate the speed-ups achieved by the $m:n$ mapping in Section 7.3.2.

#### 7.3.1 Thread Binding

Figure 7.3 shows the speed-up graphs of the benchmarks. Each graph shows the speed-ups that are achieved by one, two, four, eight, and sixteen threads for all data sizes. In addition, the ideal speed-up is depicted.

Crypt shows a speed-up of 11.7 on 16 nodes (size C, see Figure 7.3(a)). For data size A, the speed-up drops to roughly 7.8 on 16 nodes. For this data size, the effect of the DSM protocol becomes significant and deteriorates Crypt’s performance. Crypt’s kernel encrypts a sequence of bytes from one (read-only) array and writes the sequence to a second (write-only) array. As Jackal distributes the arrays in chunks of 64 KB, each thread only receives the parts of the arrays it is working on.
7.3 OpenMP/Java

Figure 7.3: Speed-ups of the OpenMP/Java benchmarks (thread binding).
However, the Jackal DSM initially allocates the input and the output array on node 0 when the benchmark starts up. Threads running on other nodes request their array chunks for reading and writing. False sharing of array chunks occurs, since two neighboring threads are likely to access the same array chunk for writing. The number of chunks that are affected by false-sharing does not exceed the number of threads. When the parallel region terminates, the DSM protocol writes back the changed array elements to node 0. This causes a serial fraction of the total runtime of the benchmark and limits its scalability.

Since Crypt would benefit if the data accessed by a thread was allocated with the respective thread, Jackal should support allocation of a single distributed array in the future.

As Figure 7.3(b) shows, Series is a pleasingly parallel benchmark. After the threads have received their fraction of the computation, no remote memory accesses are necessary for the remainder of the execution. Figure 7.3(b) shows that Series scales perfectly with the number of PEs for all data sizes. With size C a speed-up of 15.8 is achieved. For data size A the overhead of start-up and termination of the parallel region cannot be completely hidden by the long runtime of the benchmark. Thus, the speed-up drops to 14.8 on sixteen nodes. Nevertheless, JaMP achieves an optimal result for this benchmark.

SOR scales reasonably well for the large data set (see Figure 7.3(c)), because the amount of data that is shared between threads is negligible compared to the amount of the computation for each thread. Since only two rows of the grid are shared between two neighboring threads, the overhead of the remote data access is small compared to the computational effort. However, a barrier frequently synchronizes the threads after each iteration of the SOR kernel. If the barrier was not included in the kernel, SOR would achieve a speed-up of roughly 12 for sixteen nodes. With the barrier, the speed-up on sixteen nodes is limited to 11.

It is obvious that increasing the input size also improves SOR’s speed-up behavior, because the computational effort to process the matrix grows quadratically. In contrast, the DSM protocol overhead and the runtime of costs of JaMP grow linearly with the data size.

Sparse almost perfectly scales up to twelve nodes with a speed-up of roughly 9.8 (size B, see Figure 7.3(d)). For smaller node counts, Sparse even exhibits
7.3 OpenMP/Java

a super-linear speed-up. The aggregate cache of the nodes exceed the data size used for computation and thus for these node counts the benchmark is not memory-bound, but becomes CPU-bound. This is confirmed, as the super-linear speed-up disappears for the largest data set. As the number of nodes increases, the relative data size per node decreases. Hence, DSM effects and the runtime costs introduced by the parallel region become visible. For the largest data size, this effect is less significant, which is reflected in speed-ups of 7.5 on eight nodes and 13.3 on sixteen nodes.

Raytracer is a pleasingly parallel benchmark as well (see Figure 7.3(e)). A read-only data set (the spheres) is replicated to all nodes during start-up. From there on, the benchmark kernel only accesses the 1D array for writing. The only DSM protocol activity is caused when a thread accesses the LoopInformation object to retrieve the next uncomputed chunk from the work pile. The JaMP implementation is able to achieve a speed-up of roughly 15 on sixteen nodes for both data sizes.

LBM (see Figure 7.3(f)) performs a large amount of computation compared to the amount of data exchanged (see SOR). Hence, LBM shows an almost ideal speed-up behavior for the large data set (see Figure 7.3(f)).

For the small data set, however, a performance drop at fourteen nodes is observed. In LBM, each cell of the 3D grid is stored in an array that contains nine double values. To update the values of neighboring threads, the DSM protocol has to exchange 100,000 cells per LBM time step. For node counts higher than fourteen nodes, the effect of communication becomes visible in the runtime.

7.3.2 M:N Mapping

The results of the \textit{m:n} binding of JaMP are depicted in Figure 7.4. For each run of the benchmarks, the number of tasks was fixed to sixteen and the number of nodes is increased from one to sixteen.

Although mapping tasks to threads is a promising approach, the performance results are not satisfactory. As Figure 7.4 shows, some of the benchmarks show a reasonable speed-up behavior while others suffer from an unacceptable performance drop.
Figure 7.4: Speed-ups of the OpenMP-parallel benchmarks ($m:n$ mapping).
For the Crypt benchmark, Figure 7.4(a) shows a speed-up of 5.7 on sixteen nodes. In contrast, the thread binding of JaMP achieves a speed-up of 11.7 the same node count. For the smaller data sizes, the speed-up behavior is also not acceptable. A speed-up of 4.4 on data size A and 5.7 on data size C severely limits the scalability of Crypt on larger node counts. We discuss the reasons, when discussing the SOR benchmark below.

Figure 7.4(b) shows that the speed-up of Series is comparable to the speed-up achieved by the thread binding. For both bindings, the speed-up is 15.7 (data size C) on sixteen nodes. To achieve a reasonable speed-up, the number of tasks has to be a multiple of the executor threads. In this case, each executor receives the same number of task for execution. In any other case, a load imbalance is introduced, as some of the executors receive more tasks than others and thus finish their computation later.

SOR (see Figure 7.4(c)) shows an unacceptable speed-up behavior with the m:n binding. While the the threaded JaMP runtime obtains a speed-up of 13.1 on sixteen nodes, the task-based runtime stays at 4.7. The creation of the continuations to post-pone the tasks at the barrier is efficient. However, the executor service of our implementation of Java’s java.util.concurrent exhibits a large number of lock acquisitions and releases. As Jackal is forced to flush the caching heap for each acquire operation and for each release, data locality is destroyed.

Sparse also suffers from the high amount of synchronization in the executor service of Java (see Figure 7.4(d)). Again, Jackal is forced to flush the caches and data locality is destroyed. For the small data sizes, the computational effort cannot compensate the performance penalty of the cache flushes. Only for data size C, Sparse achieves a speed-up of about 10 on sixteen nodes.

Figure 7.4(f) depicts an optimal speed-up for the Raytracer benchmark. As Raytracer is a pleasingly parallel benchmark, the data locality issue does not affect the performance of the benchmark. Although, Jackal flushes the object caches, the amount of shared data is very low. Hence, the negative effect of the flushes is minimal.

LBM’s performance also suffers from the data locality problems of the m:n binding. While the JaMP runtime achieves a speed-up of 12.2 (14.9 for size B) in case of the thread binding, the speed-up drops to 8.9 and 14.1 respectively. This confirms the data locality problem, as for the large data size the performance drop is less significant. The computational effort of
Table 7.2: Overheads of JaMP’s \( m:n \) binding with respect to the thread binding.

(a) Overheads for Crypt, Series, and SOR.

<table>
<thead>
<tr>
<th>PEs</th>
<th>Crypt thread</th>
<th>Crypt task</th>
<th>Series thread</th>
<th>Series task</th>
<th>SOR thread</th>
<th>SOR task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>74,8</td>
<td>2,00%</td>
<td>2775,2</td>
<td>-1,77%</td>
<td>115,4</td>
<td>1,66%</td>
</tr>
<tr>
<td>2</td>
<td>40,5</td>
<td>2,13%</td>
<td>1366,5</td>
<td>0,07%</td>
<td>58,4</td>
<td>24,47%</td>
</tr>
<tr>
<td>4</td>
<td>22,2</td>
<td>-2,71%</td>
<td>685,4</td>
<td>0,52%</td>
<td>30,1</td>
<td>40,38%</td>
</tr>
<tr>
<td>8</td>
<td>11,2</td>
<td>16,93%</td>
<td>343,8</td>
<td>0,06%</td>
<td>16,2</td>
<td>64,47%</td>
</tr>
<tr>
<td>16</td>
<td>6,4</td>
<td>108,43%</td>
<td>175,3</td>
<td>-1,46%</td>
<td>10,5</td>
<td>138,53%</td>
</tr>
</tbody>
</table>

(b) Overheads for Sparse, Raytracer, and LBM.

<table>
<thead>
<tr>
<th>PEs</th>
<th>Sparse thread</th>
<th>Sparse task</th>
<th>Raytracer thread</th>
<th>Raytracer task</th>
<th>LBM thread</th>
<th>LBM task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>388,2</td>
<td>15,74%</td>
<td>6061,3</td>
<td>7,27%</td>
<td>685,9</td>
<td>-0,10%</td>
</tr>
<tr>
<td>2</td>
<td>198,2</td>
<td>14,27%</td>
<td>2967,3</td>
<td>16,94%</td>
<td>343,5</td>
<td>5,13%</td>
</tr>
<tr>
<td>4</td>
<td>99,2</td>
<td>26,60%</td>
<td>1492,1</td>
<td>2,30%</td>
<td>172,9</td>
<td>5,46%</td>
</tr>
<tr>
<td>8</td>
<td>51,6</td>
<td>11,36%</td>
<td>712,8</td>
<td>6,59%</td>
<td>87,3</td>
<td>5,63%</td>
</tr>
<tr>
<td>16</td>
<td>29,1</td>
<td>12,74%</td>
<td>419,2</td>
<td>6,74%</td>
<td>45,9</td>
<td>5,51%</td>
</tr>
</tbody>
</table>

the LBM kernel grows cubically with the data size, while the performance penalty only grows quadratically.

Table 7.2 confirms the above discussion of the results for the task binding of the JaMP implementation. The benchmarks that scale reasonably well (Series, Sparse, Raytracer, and LBM) exhibit a constant overhead. As the overhead is is not proportional to the number of nodes, an increased node count compensates for the performance penalty. For the other benchmarks (Crypt and SOR), the overhead grows with increasing node counts and causes a loss of performance.

The \( m:n \) binding does not rely on a fixed mapping of LUEs to SUEs, as the thread binding does. If a logical unit of execution enters a barrier, it is postponed by creating a continuation for the LUE and the executing SUE retrieves another LUE from the task queue. Hence, LUEs are likely to
move around between different SUEs during execution. Since Jackal’s thread placement policy prescribes that JaMP’s executor threads run on different nodes of the cluster, changing an LUE’s executor thread also implies moving the LUE from one node to another.

Changing the executing node of a thread in a DSM is very expensive due to the underlying network’s high latencies. Remote data accesses that stem from wrongly placed data cause severe performance penalties. All benchmarks that exhibit a low computational effort compared to the size of the data processed (i.e., Crypt, SOR, and Sparse) suffer from an additional loss of performance, as LUEs are moved between the cluster nodes during execution.

### 7.4 Reparallelization

For reparallelization, the JaMP compiler adds code to the application’s code. It inserts a reparallelization stub that tests whether or not a reparallelization has to be performed and the number of threads needs to be changed. If a change is requested, the reparallelization stub triggers the runtime to add or remove threads to retrofit the application to the new count of processing elements. Clearly, the added code adds an overhead to the application. This section investigates the performance penalty and discusses the reasons for the decreased performance.

Table 7.3 shows the overheads of the benchmarks (largest input size) with the reparallelization stubs added to the benchmark code (column “Adjustment”). As baseline we use the unmodified benchmarks, that is, the benchmark code that is augmented with the OpenMP/Java pragmas and that is compiled with JaMP’s compiler, but without enabling the reparallelization feature. We skip the evaluation of Sparse (indicated with “n/a” in Table 7.3), since Sparse introduces data dependencies to the thread ID, which we disallow (see Section 4.6). In the table, we only present the largest data size of the benchmark; smaller data sizes are discussed below.

Crypt and Series show a very low overhead if the kernels are augmented with adjustment points. The if statement that is added by the reparallelization stub does not cause a significant overhead during execution. In some cases, the performance is even increased due to branch prediction and cache effects. The same holds for LBM.
7 Performance Evaluation

Table 7.3: Overheads of adjustment points and checkpointing added.

(a) Overheads for Crypt and Series.

<table>
<thead>
<tr>
<th>PEs</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>74.8</td>
<td>1.64%</td>
<td>6.36%</td>
<td>2775.2</td>
<td>-1.21%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>40.5</td>
<td>2.37%</td>
<td>4.51%</td>
<td>1366.5</td>
<td>0.72%</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>22.2</td>
<td>4.38%</td>
<td>4.13%</td>
<td>685.4</td>
<td>-0.05%</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>11.2</td>
<td>-0.15%</td>
<td>4.09%</td>
<td>343.8</td>
<td>0.31%</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>6.4</td>
<td>-2.61%</td>
<td>4.04%</td>
<td>175.3</td>
<td>-0.74%</td>
</tr>
</tbody>
</table>

(b) Overheads for SOR and Sparse.

<table>
<thead>
<tr>
<th>PEs</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>115.4</td>
<td>0.04%</td>
<td>0.19%</td>
<td>388.2</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>58.4</td>
<td>1.96%</td>
<td>1.00%</td>
<td>198.2</td>
<td>n/a</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>30.1</td>
<td>4.95%</td>
<td>0.22%</td>
<td>99.2</td>
<td>n/a</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>16.2</td>
<td>21.33%</td>
<td>-0.50%</td>
<td>51.6</td>
<td>n/a</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>10.5</td>
<td>63.48%</td>
<td>-0.64%</td>
<td>29.1</td>
<td>n/a</td>
</tr>
</tbody>
</table>

(c) Overheads for Raytracer and LBM.

<table>
<thead>
<tr>
<th>PEs</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
<th>Baseline</th>
<th>Adjustment</th>
<th>Checkpointing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6061.3</td>
<td>4.72%</td>
<td>0.56%</td>
<td>685.9</td>
<td>0.18%</td>
<td>0.19%</td>
</tr>
<tr>
<td>2</td>
<td>2967.3</td>
<td>13.95%</td>
<td>-0.06%</td>
<td>343.5</td>
<td>0.02%</td>
<td>-0.22%</td>
</tr>
<tr>
<td>4</td>
<td>1492.1</td>
<td>10.90%</td>
<td>-2.57%</td>
<td>172.9</td>
<td>0.56%</td>
<td>0.35%</td>
</tr>
<tr>
<td>8</td>
<td>712.8</td>
<td>16.38%</td>
<td>0.16%</td>
<td>87.3</td>
<td>0.94%</td>
<td>0.17%</td>
</tr>
<tr>
<td>16</td>
<td>419.2</td>
<td>5.59%</td>
<td>-1.01%</td>
<td>45.9</td>
<td>2.78%</td>
<td>0.29%</td>
</tr>
</tbody>
</table>
Inserting the adjustment points into SOR’s kernel decreases performance by up to 63% (see Table 7.3). This unacceptable large overhead is caused by adding a constant overhead per adjustment point to a very low runtime per iteration (i.e., 584 msec with one thread, 91 msec for sixteen threads). This is confirmed by the grow of the overhead for increasing thread counts. For a single thread, the overhead is negligible and increases to 21% with eight threads. SOR is the only benchmark that shows this kind of performance degradation.

The average overhead for the support of dynamic adjustment of the thread count in parallel regions is roughly 6% for the complete set of benchmarks. If the poor performance of the SOR benchmark is not taken into account, the overhead of the dynamic adjustment drops to 3%, which we consider an acceptable overhead compared to the flexibility the application gains. Other approaches to make (MPI) applications malleable cause roughly the same overheads. For example, the malleable MPI programs of AMPI [95] cause overheads of the same order on average (about 3%).

Figure 7.5 shows the speed-ups that are achieved for the benchmarks with adjustment points added. Again, we skip the Sparse benchmark kernel, as the kernel violates the restrictions of reparallelization by introducing a dependency to the thread ID.

Table 7.3(a) shows a negligible overhead the for Crypt kernel on the large data size if adjustment points are added. The kernel with medium data size shows the same speed-up behavior as the unmodified kernel (see Figure 7.5(a)). For data size A, however, performance deteriorates as the average runtime of an iteration in the kernel’s loop is not long enough to compensate for the added overhead of the adjustment point.

The speed-up graph of SOR (see Figure 7.5(c)) confirms the result of Table 7.3(b). While the unmodified version of SOR achieves a speed-up of 11 on sixteen nodes, the overhead of the adjustment points causes a drop of the speed-up to about 6.7 on sixteen nodes. Similar results are observed for the smaller data sizes that actually exhibit a slow-down, when the node count exceeds eight nodes. Hence, although SOR is reparallelizable, a reparallelization should only be performed on large input data to compensate for the introduced overheads.

For the other benchmarks (Raytracer, Series, and LBM), the speed-up behavior of the augmented kernels closely matches the speed-ups of their un-
Figure 7.5: Speed-ups of the OpenMP-parallel benchmarks with adjustment points added.
modified counterparts. The overhead that is introduced by adding the adjustment points to the kernels is compensated by the high computational effort for the input data size.

Figure 7.6 depicts the runtime of the individual iterations of the SOR benchmark. The benchmark is started with sixteen nodes, but only one node allocates the input data and executes a single worker thread. At each 40th iteration, the JaMP runtime reconfigures the thread count and doubles the number of threads that execute the currently active parallel region. At these iterations, a runtime peak of at most 2200 msec can be observed. A small fraction of the peak attributes to the reparallelization that adds threads to the region. The bigger part of the runtime peak is caused by the DSM protocol. Any data locality is lost after the new threads have been created and start to access the matrix on the allocation node. The DSM runtime notices this and starts to re-establish data locality as fast as possible. After a few iterations, the data distribution is adapted to the new data locality needs and the kernel runs at full speed again. The performance gain closely matches the speed-up behavior of SOR in Figure 7.5(c).

We reverse the scenario for Figure 7.6(b). SOR starts with sixteen threads on sixteen nodes. After each 40th iteration, JaMP’s runtime removes half of the threads, reparallelizes the parallel region, and causes a runtime peak of at most 2400 msec. The DSM runtime notices the data locality problem and adapts the data distribution to fit the needs of the new thread count. Again, it takes ten to fifteen iterations until the data redistribution is completed and the benchmark resumes full speed. As expected, each level of performance for the different thread counts reflects the SOR’s speed-up behavior.

The same experiment is repeated for LBM (see Figure 7.7). LBM is configured to compute 100 time steps instead of 50. After each 20th time step, the thread count is doubled by the JaMP runtime system. Similar to SOR, the data locality is temporarily destroyed and the DSM runtime system has to adapt the data distribution to the new demand. In case of LBM, this causes an overly high runtime peak of about 415 sec (at time step 21). During this period of time, the runtime system redistributes eight million LBM cells (size B) between node 0 and node 1. This causes sixteen million messages on the network. At time step 41, the runtime peak is only roughly 240 sec. Although eight million cell have to be moved as well, the redistribution affects two nodes (node 0 and node 1) to scatter the cells over four nodes. Hence, more communication can be performed in parallel, which roughly halves the
7 Performance Evaluation

(a) Continuously increasing the node count by a factor of two.

(b) Continuously decreasing the node count by a factor of two.

Figure 7.6: Reparallelization of SOR (runtime per iteration).
7.4 Reparallelization

(a) Continuously increasing the node count by a factor of two.

(b) Continuously decreasing the node count by a factor of two.

Figure 7.7: Reparallelization of LBM (runtime per iteration).
height of the runtime peak. After data locality is re-established by the DSM runtime, LBM continues at full speed and shows the desired speed-ups (see Figure 7.6(b))

If the scenario is reversed (see Figure 7.7(b)), the results comply with the results of the SOR benchmark. Starting with sixteen threads on sixteen nodes, the average runtime of a single LBM time step is decreased by about a factor of two if the number of threads is continuously halved at each 20th time step. Similar to the scale-up experiment, each reparallelization indirectly causes a runtime peak of about 450 sec that is almost completely attributable to the destroyed data locality and the DSM runtime. The DSM runtime needs up to six iterations of the time-step loop to re-establish the data distribution needed. After the DSM runtime has finished data redistribution, LBM runs at full speed and does not suffer from additional performance penalties.

As Section 7.6 will show, the overhead of data redistribution does not affect program execution as much as expected from the artificial benchmark setting. The times needed for checkpointing and waiting in the cluster’s queues are much higher, effectively mitigating the reparallelization costs.

### 7.5 Checkpointing

To enable checkpointing for an application, the compiler adds invocations to `createCheckpoint` to the application’s code. If no checkpoint is requested, `createCheckpoint` immediately returns and performs no further actions. Only if a checkpoint creation is currently requested, the application notices a performance impact, as it is blocked until the checkpoint has been created. During normal operation, however, the impact on application performance should be minimal, as the Jackal compiler places the invocations of `createCheckpoint` in a sophisticated way.

Table 7.3 confirms this assumption. The column “Checkpointing” shows the overheads that are caused by the added `createCheckpoint` calls. The average performance impact of our checkpointing package is 1%. However, there are benchmarks with tight loops for which the overhead exceeds 2%. Specifically, Crypt with the low iteration runtime suffers from an overly high performance penalty of at most 6%. Although it is compute-intensive, Series also contains a very tight loop with short iteration times. Our checkpointing package adds an overhead of 2% to the baseline execution.
7.5 Checkpointing

Figure 7.8: Speed-ups of the OpenMP/Java benchmarks with checkpoint support added.
Table 7.4: Checkpoint sizes (in MB) of the benchmarks (largest data size).

<table>
<thead>
<tr>
<th># PEs</th>
<th>Crypt</th>
<th>Series</th>
<th>SOR</th>
<th>Sparse</th>
<th>Raytracer</th>
<th>LBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>116</td>
<td>&lt;1</td>
<td>15</td>
<td>171</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>2</td>
<td>116</td>
<td>&lt;1</td>
<td>15</td>
<td>171</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>4</td>
<td>116</td>
<td>&lt;1</td>
<td>15</td>
<td>171</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>8</td>
<td>116</td>
<td>&lt;1</td>
<td>15</td>
<td>171</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>16</td>
<td>117</td>
<td>&lt;1</td>
<td>15</td>
<td>171</td>
<td>1</td>
<td>541</td>
</tr>
</tbody>
</table>

The speed-up graphs of Figure 7.8 show that the augmented code does not limit the speed-up of the individual benchmarks. As the `createCheckpoint` calls only cause an overhead of 1% on average and the overhead does not increase with the node count, no benchmark suffers from a performance degradation.

Table 7.4 shows the size of the checkpoints that are created for the largest data size of each benchmark. The checkpoint sizes are unrelated to the number of nodes, as the total amount of data is determined by the benchmark size and is not changed with the node count. Thus, for each configuration the checkpointing algorithm needs to write to disk the same amount of data that is distributed among the nodes.

The effect of compression lowers the size of the checkpoints by a considerable amount. For example, Crypt processes 200 MB of data and only consumes 116 MB in the checkpoint. This corresponds to a compression rate of over 42%. For SOR the compression rate is even higher and amounts to almost 95%. For LBM, the compressed checkpoint consumes roughly 25% of the total amount of data in the application heap. The time needed to write LBM’s heap (2.3 GB of data) to disk is about 37 sec, which corresponds to a used disk bandwidth of roughly 14.5 MB/sec.

### 7.6 Migration

For the migration experiment, we use the Woodcrest cluster of the Friedrich-Alexander-University, Germany, as the migration source. As migration target, we use the DAS3 cluster of the Vrije Universiteit in Amsterdam, The Netherlands.
7.6 Migration

The DAS3 cluster consists of quad-core AMD Opteron nodes (2×2 cores) with a clock frequency of 2.4 GHz. Each machine is equipped with 4 GB of main memory and runs Scientific Linux (kernel version 2.6.18-clustervision-136.1_cvos). The cluster nodes offer Myrinet 10G and Gigabit Ethernet as interconnection networks.

On the front-end nodes of both clusters, we started OGRE instances that connect to each other through the P2P network. LBM was configured to execute 1,000 time steps on a $200 \times 200 \times 200$ grid. To obtain a reasonable fast execution of LBM, we restricted the execution to the fast development queues of both clusters. The development queues allow users to quickly gain access to cluster nodes, as the maximum permissible walltime for a job is limited. On Woodcrest, the walltime limit for this queue is one hour; on DAS3, the limit is fifteen minutes. Figure 7.9 shows the result of the experiment and depicts the runtime of individual LBM time steps during the execution of LBM on the different clusters.

We submitted LBM at the OGRE daemon running on the Woodcrest cluster. The daemon requested bids from all clusters and decided to migrate the application to the DAS3 cluster. The limit of fifteen minutes was exploited to compute roughly 140 time steps on the grid. The daemon then caused a checkpoint creation, aborted LBM, and moved it to the Woodcrest cluster. The transfer of 540 MB of checkpoint data took about 122 sec, which

![Figure 7.9: Continuous migration of LBM between Woodcrest and DAS3.](image-url)
corresponds to a bandwidth of roughly 4.5 MB/sec. Without compression, the transfer of the checkpoint would have needed about 511 sec. This proves that compressing checkpoints effectively lowers the costs that account to migration.

The reparallelization runtime was informed to adapt the degree of parallelism of the LBM kernel. The runtime peak attributes to the time needed to create and transfer the checkpoint, wait for the job to start on the target system, and re-establish data locality.

At the Woodcrest cluster, LBM resumed from the checkpoint and continued with just two nodes for roughly an hour. This corresponds to about 450 time steps. After an hour, LBM was checkpointed again and migrated to a local reservation with four nodes to speed-up computation of the next 330 time steps. During this migration no costs for copying the checkpoint between the systems incurred.

As LBM did not finish computation on time, the OGRE daemon decided to send LBM back to The Netherlands to continue computation after the second reservation at the Woodcrest cluster. OGRE claimed sixteen nodes and restarted LBM. On the DAS3 cluster, LBM computed the last time step and successfully terminated. The total runtime of the experiment was about 120 minutes.

7.7 Summary

With a set of microbenchmarks and application benchmarks, we have evaluated the performance of our reparallelization and migration solution.

The microbenchmarks show that our implementation of OpenMP/Java is efficient and does not introduce unnecessary overheads. The largest fraction of the runtime costs of the OpenMP/Java constructs attributes to the Jackal DSM protocol and is unrelated to the implementation of JaMP. We have also compared the costs caused by static and dynamic scheduling for parallel loops.

The application benchmarks show the expected speed-up behavior for the JaMP thread binding. The pleasingly parallel benchmarks Series and Raytracer show optimal speed-ups when compiled for the thread binding of the
7.7 Summary

JaMP runtime. All other benchmarks exhibit a reasonable speed-up behavior that closely matches the scalability of the benchmark kernel.

While the task-based $m:n$ binding seems to be a promising approach to compile OpenMP/Java programs, the evaluation showed that not all benchmarks are suitable to be executed with an $m:n$ mapping of OpenMP threads to executors. Benchmarks that execute a tight loop (e.g. Crypt and Sparse) do not obtain reasonable speed-ups. The tight loop cannot compensate the performance penalty that is introduced by lost data locality due to the scheduling overhead of the executor service. Pleasingly parallel benchmarks (Series), however, benefit from the increased node count, as their data set is too small to exhibit data locality effects.

The code that is added by our JaMP compiler to enable reparallelization for an application, introduces an overhead of roughly 6% at maximum. If the pathological case of SOR is ignored, the overhead drops to 3%. SOR’s performance degrades as the overhead of an adjustment point cannot be compensated by the short iteration time of a single loop in the SOR kernel. We have tested continuous reparallelization with the SOR and LBM benchmark. Both show the desired behavior if the thread count is increased or decreased. A runtime peak is caused, as during reparallelization data locality is temporarily lost and has to be re-established by the DSM runtime. Thereafter, the benchmarks continue at full speed.

Augmenting the application code with `createCheckpoint` invocations does not hurt the performance characteristics of the benchmarks. Support for our checkpointing package decreases performance by roughly 1% on average. Thus, the overhead of our checkpointing solution is almost negligible. The checkpointing thread compresses the checkpoint data by at least 42% (at most 94%) and writes to disk with a speed of roughly 14.5 MB/sec.

We have migrated LBM between two clusters in Germany and The Netherlands to prove the feasibility of our migration framework. The job has been started in Germany and has been smoothly migrated to The Netherlands, and back. This was repeated until LBM successfully finished its computation and terminated.
8 Conclusions and Future Work

8.1 Conclusions

We have presented a novel approach to reparallelize and to migrate OpenMP applications between clusters of different size and architecture. The solutions consists of the following components.

We have defined an OpenMP binding for the Java programming language. With its high-level declarative syntax, OpenMP provides a convenient programming model for loop-based parallelism. The programmer is relieved from the need to handle low-level multi-threading and to cope with technical details such as thread creation, work distribution, and synchronization. Our OpenMP/Java specification is the first successful attempt to address all issues of an integration of the OpenMP programming model and the Java programming philosophy. We fully adopt the OpenMP syntax and semantics. We have extended the OpenMP semantics achieve a tighter integration into the Java’s object-oriented programming paradigm, where necessary.

We have presented an OpenMP/Java implementation for the Jackal compiler and DSM system. JaMP supports two different compilation schemes: multi-threaded binding and \( m:n \) task binding. The JaMP thread binding employs state-of-the-art compilation techniques to compile OpenMP/Java codes into regular multi-threaded code that uses the Java threading API. OpenMP threads are mapped to executor threads in a 1:1 fashion, that is, each OpenMP thread corresponds to exactly one executor thread. The \( m:n \) task binding decouples OpenMP threads and executor threads. In this binding, \( m \) OpenMP threads are multiplexed on \( n \) executor threads. We designed the JaMP runtime to be flexible and efficient at the same time.
With reparallelization, OpenMP applications can transparently alter their degree of parallelism while a parallel region is actively executing. The OpenMP compiler of JaMP augments the application code with so-called adjustment points at which threads can be removed from or added to the execution of the parallel region. Adjustment points are compiled to `if` statements and runtime system calls that trigger the reparallelization. If threads are added or removed, the runtime system automatically adapts the data and work distribution of parallel regions to the new thread count. All OpenMP constructs supported are effectively reparallelized without introducing unacceptable overheads (roughly 3% on average).

Our migration solution is based on a near overhead-free checkpointing package. The checkpointing package stores the computational state of an application to disk. The state can then be transferred to a new reservation on the local system or to a reservation on a remote system. For migration, the application must be resumed on various different architecture. We have solved the problem of creating platform-independent checkpoints without introducing noticeable overheads. During normal operation, the application’s performance is decreased by at most 2%. Our compiler adds `createCheckpoints` invocations to the application code. The foundation of our checkpointing package are checkpointer and uncheckpointer functions that save and restore the state of live variables at the `createCheckpoints` into a platform-independent format. The stack is unwound to recursively save the state of all methods on the call stack; the application heap is saved by means of standard Java serialization and compression.

With OGRE, we have designed a prototype implementation of a migration framework that automates the process of application migration. On each accessible cluster an OGRE daemon is started. The daemon searches for free resources and triggers migrations of the application. We have defined a flexible architecture that enables researchers to freely adapt the internal control flows of the framework. As a proof of concept, we have implemented a default migration strategy that uses a simplistic performance model to describe the computing power of a cluster.

Our evaluation with a set of microbenchmarks and application-kernel benchmarks show that the JaMP runtime is efficiently implemented. All overheads of the runtime attribute to Jackal’s DSM protocol. We have compared the thread binding and task-based $m:n$ binding of JaMP with the application-kernel benchmarks. For the thread binding, these benchmarks show a good
speed-up behavior and closely match the expected speed-up for each kernel. The \( m:n \) binding suffers from increased synchronization overheads and thus does not achieve comparable results. The reparallelization adds an overhead of roughly 3\% to the kernels and is able to successfully adapt the degree of parallelism of two kernels. Runtime peaks appear during reparallelization, since the Jackal DSM needs to redistribute data to re-establish data locality for the kernels. We have shown that our migration framework is able to successfully transfer a compute-intensive application between clusters. The application was smoothly migrated between two clusters located in Germany and The Netherlands.

We have shown that it is possible to automatically alter the degree of parallelism of OpenMP programs while they are actively executing a parallel region. Our solution is completely transparent to the programmer, as the compiler automatically augments the unmodified OpenMP code to provide the reparallelization features. The overhead of the added code is negligible compared to the added flexibility of freely adapting the degree of parallelism. Our transparent migration feature also does not impose unacceptable overheads on the application's performance. The compiler automatically adds the checkpointing code to the binary, which decreases performance by 1\% only. The application gains the flexibility to be executed on any cluster of the computational Grid. Clusters that offer free computing resources become transparently available to the application without the need to change the application code. The prototype of the migration framework successfully migrates applications between clusters.

Combined to a flexible system for application migration, all of the above mentioned components enable an application to freely migrate between clusters of different size and architecture. Our solution helps to blur the boundaries of individual clusters of a computational Grid and support the user to utilize the available computing power more easily. A user can start an application at virtually any point of the Grid infrastructure and our system transparently searches for free computing resources. The application automatically claims the free resources and migrates to continue computation at the claimed resources. Users are no longer forced to estimate the runtime of their jobs, as the application automatically approximate their runtime in a piecewise fashion. In addition, the cluster scheduler does not necessarily suffer from reservation holes, as these holes can be filled with migratable applications.
8.2 Future Work

We have shown techniques for reparallelization and migration. These techniques may contribute to many aspects of multi-threaded programming and utilization of computing systems. In the following, we will quickly depict future extensions to the presented solutions to make use of them in different areas. First, we present a set of technical issues that could be resolved in the future. We then discuss adaptive scheduling on multi-core systems, adaptive load balancing of parallel loops, and fair-share scheduling on clusters.

Unresolved technical issues

The current specification of OpenMP/Java is based on the OpenMP binding for C++ and only supports the constructs and directives of version 2.5. As version 3.0 of the OpenMP specification is available, a future extension of OpenMP/Java should consider adding the missing constructions and directives. Especially the `task` construct might prove useful in future versions of OpenMP/Java.

As the JaMP implementation is based on the OpenMP/Java specification presented in this work, JaMP also only supports the constructs and directives that are available in OpenMP 2.5. If the specification of OpenMP/Java is extended with new features, the implementation of JaMP should be extended as well. For larger thread counts, the JaMP implementation should support different algorithms to construct the thread team and to perform reductions. For example, a tree-based reduction algorithm might reduce the linearly growing runtime costs of reductions. A future version should also replace the barrier implementation of Jackal with a specialized fast barrier for JaMP applications.

During a reparallelization on NUMA or COMA systems, data locality is destroyed if threads are added or removed. Our current implementation relies on the computing environment (i.e., the Jackal DSM) to automatically re-establish data locality after the reparallelization has been performed. Data redistribution could be applied to the data locality problem. As the compiler knows about its templates for generating the parallel code, it could analyze the parallel region and deduce the data partitioning. If a reparallelization is triggered, the compiler-generated code for this region could redistribute the data accordingly and re-establish data locality after a reparallelization.
We have also presented a prototype implementation of a migration framework. OGRE is able to successfully migrate an application between clusters. However, it only uses a very simplistic performance model to assess the computing power of the target clusters. Moreover, the default migration strategy could be improved to better estimate the benefits and costs of a migration. Currently, OGRE only support the cluster management systems of two clusters (i.e., Woodcrest and DAS3). New batch system communicators for other cluster management systems should be added to make OGRE available for the majority of cluster systems.

**Adaptive scheduling in operating systems**

Operating system schedulers currently cannot influence the degree of parallelism of applications. While most applications nowadays are single-threaded applications, this limitation is not a severe restriction. However, with the advent of multi-core CPUs, the fraction of parallel applications will increase to make use of the additional cores on the chip. In the future, the scheduler of the OS will be faced with a large number of application-level threads that have to be scheduled for execution on the cores of the CPUs in the system. Since applications are typically not aware of the other applications running on the same system, the applications are likely to spawn threads without considering the number of threads already spawned by other programs.

Currently, the scheduler does not have any means to influence the running applications’ degree of parallelism. It is desirable to change the overall number of threads executed according to the system utilization. For example, a parallel application that starts memory-bound threads is expected to not obtain good speed-ups when its threads are scheduled to run on a single chip. Each of the cores accesses the same memory path and thus the bandwidth available for each core drops to a fraction of the total bandwidth only. In such cases, the operating system scheduler could cause a reparallelization of the application and remove threads from it.

If applications could specify a lower bound and an upper bound for their degree of parallelism, the scheduler could be enabled to choose the degree of parallelism of each individual application such that the resources of the system are used optimally. The scheduler could attempt to find a trade-off between the applications’ specifications and system utilization. As new
applications are started and terminated at any time, the scheduler must continuously retrofit the degrees of parallelism of running applications to the new demand. Reparallelization is important, since it provides the means to adapt the applications' parallelism to the new requirements.

Although reparallelization is a reasonably cheap operation, it causes data locality issues that attribute for the largest fraction of the costs of a reparallelization. Hence, the scheduler should only trigger a reparallelization if the costs for reparallelization are compensated by the performance gain for the system. As the compiler has access to the source code and is able to analyze the program, a compiler could provide the scheduler with the needed information. The compiler could insert calls to the scheduler API to inform the scheduler about the expected amount of data for which data locality is violated. The scheduler may then estimate the benefit and trigger a reparallelization if it seems beneficial.

It is also possible to move the code that performs reparallelization of parallel regions into the operating system and only augment the application with adjustment points that trigger the OS if a reparallelization is requested. As the operating system has full access to the application’s memory image, the operating system could exploit processor traps to stop the application if a reparallelization is required. With processor traps, even the adjustment points could be removed from the application and moved into the OS kernel. At the former adjustment points, the OS could place instructions that trigger the processor trap and perform the reparallelization. Hence, the overheads of the added code could be avoided.

Adaptive load balancing of parallel loops

Parallel loops in loop-based programs are seldom perfectly balanced. In most cases, a subset of the iterations of the loop are computational intensive, whereas the remainder only constitutes a small fraction of the parallel work. If such loops are scheduled in a static way (e.g. by adding \texttt{static} to a work-sharing clause), a load imbalance is introduced. As each thread receives the same number of iterations, some threads are overloaded, while others are undersupplied with work. Although a programmer can overcome this problem by using the \texttt{dynamic} clause at the work-sharing construct, it
is cumbersome and unsatisfactory to search for the optimal chunk size depending on the input data, the architecture of the machine, and the current load of the system.

An automatic means to determine the load of the loop at runtime is desired: The runtime system should monitor the execution of parallel loops and detect load imbalances. If an imbalanced parallel loop is detected, the reparallelization runtime could be instructed to remove undersupplied threads that have been assigned to less intensive loop iterations. The reparallelization runtime then could reassign these threads to other areas of the iteration space to support overloaded threads by assigning compute-intensive iterations to undersupplied threads.

The compiler and the runtime system could enhance adjustment points to support adaptive load balancing. Adjustment points already query a strategy object that decides which threads are scheduled for removal. It also determines how many threads have to be added if the thread count is increased. This strategy object can be reused to implement a corresponding feature. However, the compiler has to add monitoring code that profiles the execution of the loop. It needs to augment the adjustment point with code to measure the performance (e.g., the execution time) of single loop iterations and pass the data to the reparallelization runtime. The reparallelization runtime could evaluate the performance data and inform the strategy object whether or not threads must be reassigned to other fractions of the loop’s iteration space.

**Fair-share scheduling on clusters**

Reparallelization and migration try to minimize the number of reservation holes of a cluster. Reparallelization adapts an application’s parallelism to fit the number of nodes of the reservation hole. Migration saves the application from premature termination and transfers the application to a new cluster reservation. This is required because of two reasons. First, a scheduler of a cluster typically operates in batch mode, that is, jobs (statically) request a prescribed number of nodes for a certain amount of time. Second, users overestimate the runtime of jobs to avoid premature termination.

With fair-share scheduling, the jobs should not request a certain number of nodes, but instead could be launched on a cluster for processing without
8.2 Future Work

specifying resource limits. A hypothetical cluster scheduler is responsible for assigning the number of nodes to the currently active applications. According to the priority of the jobs and the users’ fair-share value (i.e., the percentage of the cluster a user may occupy on average) the scheduler could automatically adapt the degree of parallelism of all running jobs.

If new jobs were to be started on the cluster, the scheduler should reparallelize the existing jobs to release nodes that can then be utilized by the started application. Conversely, the scheduler may assign the nodes of a terminated application to the still running applications to speed-up their execution.

Similar to the adaptive scheduling in operating systems (see above), the costs of a reparallelization must be taken into account before changing the degree of parallelism for any of the running jobs. If more than one job qualifies for reparallelization, the scheduler should choose the job with the least effort of reparallelization, that is, the job that experiences the smallest performance penalty. In addition, creating a process on a remote machine is a costly operation. Lost data locality also imposes a much higher cost on cluster systems than on shared-memory machines. A cost model should reflect these costs; the scheduler then could use the cost model to decide which jobs should be reparallelized and which job is left as is.

Migration gives another degree of freedom to the scheduler. If, for example, a job with a high priority starts to execute on the cluster, the scheduler could preempt one or all of the existing jobs to make nodes available to the highly prioritized job. As soon as the prioritized job terminates, the postponed jobs are resumed and continue operation as normal.

Since with reparallelization and migration the scheduler gains other degrees of freedom to schedule applications on the system, creating the schedule becomes a more complex operation. Hence, the computational effort finding an optimal schedule is likely to be increased. Future work should investigate the potential of new scheduling algorithms that take into account reparallelization and migration and that create an optimal fair-share schedule for application on clusters.


Bibliography


255


Bibliography


Bibliography


Bibliography


Bibliography


List of Acronyms

ABI ............... Application Binary Interface
API ............... Application Programming Interface
AST ............... Abstract Syntax Tree
CFG ............... Control Flow Graph
CLOMP ........... Intel Cluster OpenMP
COMA ............. Cache Only Memory Architecture
COOL ............. Concurrent Object-Oriented Language
CPU ............... Central Processing Unit
DAG ............... Directed Acyclic Graph
DFA ............... Data-Flow Analysis
DJVM ............. Distributed Java Virtual Machine
FIFO ............. First-In First-Out
GOR ............... Global Object Reference
HPC ............... High Performance Computing
HPF ............... High Performance Fortran
ICV ............... Internal Control Variable
JGF ............... Java Grande Forum
JIT ............... Just In Time
JLS ............... Java Language Specification
JMM ............... Java Memory Model
JNI ............... Java Native Interface
JVM ............... Java Virtual Machine
LBM ............... Lattice-Boltzmann Method
LUE ............... Logical Unit of Execution
MCM ............... Memory Consistency Model
MMU ............... Memory Management Unit
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTBF</td>
<td>Mean Time Between Failure</td>
</tr>
<tr>
<td>NOW</td>
<td>Network of Workstations</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OGRE</td>
<td>Open Grid Research Environment</td>
</tr>
<tr>
<td>OGSI</td>
<td>Open Grid Services Infrastructure</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer to Peer</td>
</tr>
<tr>
<td>PCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>SC</td>
<td>Sequential Consistency</td>
</tr>
<tr>
<td>SESE</td>
<td>Single-Entry Single-Exit</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processor</td>
</tr>
<tr>
<td>SUE</td>
<td>System Unit of Execution</td>
</tr>
<tr>
<td>TBB</td>
<td>Threading Building Blocks</td>
</tr>
<tr>
<td>UDS</td>
<td>Unified Descriptor Strings</td>
</tr>
<tr>
<td>UE</td>
<td>Unit of Execution</td>
</tr>
<tr>
<td>UPC</td>
<td>Unified Parallel C</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WC</td>
<td>Weak Consistency</td>
</tr>
</tbody>
</table>