JavaGrande: Hochleistungsrechnen mit Java

Michael Philippsen

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5, 76128 Karlsruhe
phlipp@ira.uka.de

Zusammenfassung Das *JavaGrande-Forum* ist ein Zusammenschluß derjenigen Anwender, Forscher und Firmenvertreter, die entweder versuchen, ressourcenintensive Anwendungen mit Hilfe der Programmiersprache Java zu realisieren, oder die versuchen, die Java-Programmierumgebung dahin zu entwickeln, daß es überhaupt möglich wird, „große“ Anwendungen mit Hilfe von Java effizient zu realisieren.
Wesentliche bisherige Beiträge des JavaGrande-Forums sind die Schlüsselworte `strictfp` und `fastfp` für eine verbesserte Fließkomma-Arithmetik, die Arbeiten auf dem Gebiet der komplexen Zahlen und der mehrdimensionalen Felder, sowie die schnelle Serialisierung und das schnelle RMI und schließlich die Benchmark-Initiative.
Durch seine Arbeit hat das JavaGrande-Forum international und auch bei Sun Microsystems Beachtung gefunden. Es wurde ein neuer Forschungszweig eröffnet, dessen Ergebnisse Eingang in die Fortentwicklung der Programmiersprache Java und ihrer Laufzeitumgebung fanden und finden.

1 JavaGrande-Forum

Inspiriert durch den Kaffeehausjargon, etablierte sich in letzter Zeit das Schlagwort der *Grande*-Anwendungen.[1] Solche lassen sich sowohl durch ihre Art als auch durch ihre Anforderungen an die Rechnerinfrastruktur charakterisieren. Eine Grande-Anwendung ist im wissenschaftlichen, ingenieurmäßigen oder kommerziellen Rechnen angesiedelt und zeichnet sich durch komplexe rechen-, daten- und/oder ein/ausgabeintensive Bearbeitungsschritte aus. Typische Anwendungsklassen sind Modellbildung, Simulation oder Datenauswertung. Zur Bewältigung der Bearbeitungsschritte erfordert eine Grande-Anwendung in jedem Fall sehr hoche Rechenleistung, eventuell sogar Parallelität oder verteiltes Rechnen.

Das *JavaGrande-Forum* [20] ist ein Zusammenschluß derjenigen Anwender, Forscher und Firmenvertreter, die entweder versuchen, Grande-Anwendungen mit Hilfe der Programmiersprache Java [14] zu realisieren, oder die versuchen, die Java-Programmierumgebung so zu erweitern oder zu verbessern, daß es überhaupt möglich wird, Grande-Anwendungen mit Java effizient zu realisieren.

[1] Grande, wie in Rio Grande, ist in mehreren Sprachen die übliche Bezeichnung für groß. Im Amerikanischen hat sich „grande“ auch als Größenangabe bei Bestellungen in Kaffeehäusern etabliert.

Das JavaGrande-Forum wurde im März 1998 in einer „Birds-of-a-Feather“-Sitzung in Palo Alto gegründet. Es ist ein offenes Forum, an dem jeder Interessierte ohne Mitgliedsbeiträge teilnehmen kann. Neben einer Web-Seite [20] und einer Mailingliste [21] hat das JavaGrande-Forum bzw. haben dessen Arbeitsgruppen in der Vergangenheit regelmäßig Gruppen-Treffen veranstaltet. Als wissenschaftlicher Koordinator fungiert Geoffrey C. Fox (Syracuse), als zentraler Kontaktmann bei Sun Microsystems fungiert Sia Zadeh.

Ziele des JavaGrande-Forums

Die wesentlichen drei Ziele des JavaGrande-Forums sind:

1. Evaluation der Verwendbarkeit der Programmiersprache Java und ihrer Laufzeitumgebung für Grande-Anwendungen
2. Zusammenführung der „Java Grande-Gemeinde“, Erarbeitung eines konsensfähigen Anforderungskatalogs und Bündelung der Interessen gegenüber Sun Microsystems (oder gegenüber einem eventuell künftig zuständigen internationalen Standardisierungskommitee)
3. Erarbeitung von konsensfähigen Prototypimplementierungen, Schnittstellen (APIs) und Änderungsvorschlägen, die die Programmiersprache Java und ihre Laufzeitumgebung für Grande-Applikationen verwendbar machen

Mitglieder des JavaGrande-Forums

Am JavaGrande-Forum beteiligen sich Firmen, Forschungseinrichtungen und Labors, vorwiegend aus den USA, wobei in letzter Zeit auch in Europa JavaGrande Aktivitäten zu beobachten sind [22].

Neben Sun Microsystems beteiligen sich IBM und Intel, sowie Least Square, MathWorks, NAG, MPI Software Technologies und Visual Numerics, um nur einige zu nennen. Gerade für Fragen des schnellen numerischen Rechnens ist die Zusammenarbeit mit den Hardware-Herstellern von zentraler Bedeutung.

Aus der akademischen Welt sind unter anderen die Universitäten aus Chicago, Syracuse, Delft, Berkeley, Houston, Karlsruhe, Tennessee, Chapel Hill, Edinburgh, Westminster und Santa Barbara zu nennen.

Ferner wirken das amerikanische Institut für Standardisierung (NIST), die Sandia Labs und ICASE mit.

Die Mitglieder organisieren sich in zwei Arbeitsgruppen. Abschnitt 3 beschreibt die Ergebnisse der Arbeitsgruppe Numerisches Rechnen. Abschnitt 4 gibt die Ergebnisse der Arbeitsgruppe Parallelismus und Verteilung wider.

Die bisherigen Ergebnisse und Aktivitäten des JavaGrande-Forums sind von Sun Microsystems gut aufgenommen worden. Gosling, Lindhom, Joy und Steele haben sich intensiv mit den Arbeiten des JavaGrande-Forums auseinandergesetzt bzw. für deren Umsetzung innerhalb von Sun gesorgt. Für eindrucksvolle Öffentlichkeitswirkung sorgte der Panel-Beitrag von Bill Joy, der vor knapp 21.000 Zuhöhrern der JavaOne 1999 die herausragenden Arbeiten des JavaGrande-Forums anerkennend lobte.

Wissenschaftliche Aktivitäten des JavaGrande-Forums

Neben dedizierten Arbeitstreffen ist das JavaGrande-Forum seit seinem Bestehen bemüht, bei wissenschaftlichen Konferenzen mit oder in Panels vertreten zu sein. Ferner veranstalten Mitglieder des JavaGrande-Forums Workshops. Die zentrale Veranstaltung des Forums ist die ACM Java Grande Conference, die der wissenschaftlichen Arbeit des Forums den formellen Rahmen gibt. Die wissenschaftliche Arbeit ist wichtig, um die „JavaGrande-Gemeinde“ zusammenzubringen bzw. zusammenzuhalten. Nur dann ist es möglich konsensfähige Ideen zu erarbeiten und gebündelt – und damit mit höherer Durchsetzbarkeit – gegenüber Sun Microsystems zu vertreten. Dies ist umso wichtiger je weniger Sun Microsystems die Urheberrechte an Java an internationale Standardisierungsgremien abgibt.

Bisher fanden folgende Veranstaltungen statt, bzw. sind in der Vorbereitung:

- Workshop, Syracuse, Dezember 1996, [10]
- PLDI Workshop, Las Vegas, Juni 1997, [11]
- Java Grande Konferenz, Palo Alto, Februar 1998, [12]
- EuroPar Workshop, Southampton, September 1998, [9]
- Supercomputing, Ausstellung und Panel, Orlando, November 1998, [38]
- IEEE Frontiers 1999 Konferenz, Annapolis, Februar 1999
- HPCN, Amsterdam, April 1999, [37]
- IPPS/SPDP, San Juan, April 1999, [8]
- SIAM Meeting, Atlanta, Mai 1999
- IFIP Working Group 2.5 Meeting, Mai 1999
- Mannheim Supercomputing Konferenz, Juni 1999
- ACM Java Grande Konferenz, San Francisco, Juni 1999, [13]
- JavaOne, Ausstellung und Panel, San Francisco, Juni 1999, [39]
- ICS'99 Workshop, Rhodos, Juni 1999
- Supercomputing, Ausstellung und Panel, Portland, September 1999
- ISCOPE, JavaGrande Tag, Dezember 1999, [36]

- IPPS, Cancun, Mai 2000
- Dagstuhl Seminar, August 2000
- ACM Java Grande Konferenz, San Francisco, Juni 2000

Ein Großteil der wissenschaftlichen Arbeiten der „Java Grande-Gemeinde“ sind in mehreren Ausgaben der Zeitschrift *Concurrency – Practice & Experience* dokumentiert [10–13]. Ein weiterer Teil ist in den oben genannten Tagungsbänden enthalten. Ferner gibt das JavaGrande-Forum in regelmäßigen Abständen Arbeitsberichte heraus [38, 39].

2 Warum Java?

Es gibt eine ganze Reihe von Gründen, aus denen es sinnvoll erscheint, Grande-Anwendungen mit Hilfe der Programmiersprache Java zu realisieren. Neben

den üblichen Gründen, die auch im Fall „gewöhnlicher“ Anwendungen für Java sprechen, wie z.B. der Portabilität, der Existenz von Entwicklungsumgebungen, dem (vermeintlich) produktivitätssteigernden Sprachentwurf (mit automatischer Speicherbereinigung und Thread-Unterstützung) und der Existenz einer reichhaltigen Standardbibliothek, gibt es für Grande-Anwendungen weitere wichtige Aspekte.

Das JavaGrande-Forum beurteilt Java als universelle Sprache, mit der es möglich ist, das ganze Spektrum der Aufgaben zu bewältigen, die bei Grande-Anwendungen anfallen. Zum einen bietet Java eine standardisierte Infrastruktur an, mit der sich auch in heterogenen Rechnerkonstellationen verteilte Anwendungen realisieren lassen. Die graphische Oberfläche ist gut integriert (wenn auch mit einigen Schwierigkeiten hinsichtlich der Portabilität) – ein Aspekt, der gerade für die Visualisierung von Grande-Daten ein Vorteil gegenüber anderen Programmiersprachen ist. Neben diesen klassischen Frontend-Aufgaben, kann Java in jedem Fall als Verbindungs-Code verwendet werden, um dedizierte existierende Hochleistungsanwendungen zu koppeln, um Berechnungen miteinander zu verbinden, die in anderen Programmiersprachen realisiert sind, und um als universelle Zwischenschicht zwischen Berechnung und Ein/Ausgabe zu dienen. Schließlich deuten Experimente darauf hin, daß Java auch schnell genug sein kann, um die zugrundeliegende Funktionalität von Grande-Anwendungen selbst zu realisieren.

Ein weiterer nicht unbedeutender Aspekt für die Anwender aus Wissenschaft und Ingenieurwesen ist, daß Java gelehrt und begeistert gelernt wird und schon jetzt eine breite Akzptanz hat – also daß Java Eigenschaften hat, die Fortran mehr und mehr fehlen und dadurch in absehbarer Zeit bei Fortran zu ernstem Nachwuchsmangel führen wird.

Ablaufgeschwindigkeit und Speicherbedarf

Das am weitesten verbreitete und zugleich hartnäckigste Gerücht ist, daß Java-Programme extrem langsam laufen. Die allererste öffentlich verfügbare Version von Java (1.0 beta) hat ByteCode wirklich interpretiert und war in der Tat extrem langsam. Seitdem hat sich aber viel verändert. Praktisch kein Java-Programm wird mehr rein interpretativ ausgeführt. Stattdessen werden sogenannte Just-In-Time-Übersetzer unterschiedlichster Prägung verwendet, um entweder vorab oder ausführungsbegleitend durch Übersetzung die Ausführungszeit zu verbessern.

Die folgenden drei Beobachtungen sollen dazu dienen, durch eine solidere Beurteilungsbasis den Gerüchten ihre Basis zu entziehen.

1. **Laufzeitunterschiede in der Größenordnung der Personalqualität.** Prechelt [34] führte ein Experiment an der Universität Karlsruhe durch, bei dem das gleiche Problem mehrfach, von insgesamt 38 verschiedenen Personen gelöst wurde. Dabei entstanden 24 Java-Implementierungen, 11 C++-Implementierungen und 5 Realisierungen in C. Unter den 38 Personen waren sehr gute bis relativ schlechte Programmierer.

Die Experimentteilnehmer, die durchweg Informatik-Studenten im Hauptdiplom mit durchschnittlich 8 Jahren und 100 KLOC Programmiererfahrung waren, sollten ein möglichst zuverlässiges Programm schreiben; das Experiment war nicht als Geschwindigkeitswettbewerb ausgelegt.

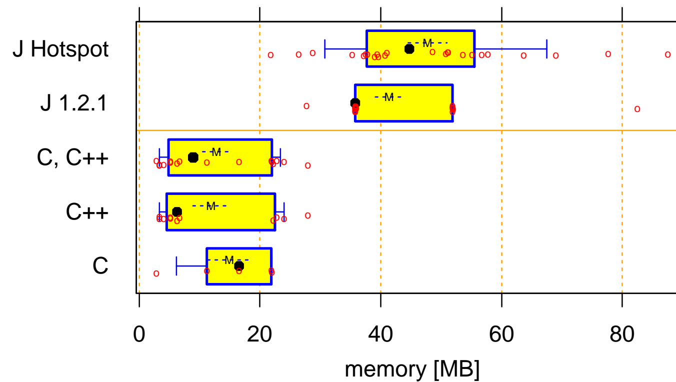

Abbildung1. Auf Solaris 7 gemessener Hauptspeicherbedarf der untersuchten Programme (einschließlich Datenstrukturen, Programmcode, Bibliotheken, Prozeßverwaltung und JVM). Die einzelnen Messungen sind durch kleine Kreise symbolisiert. Das M zeigt jeweils den Mittelwert, der schwarze Punkt gibt den Median an. Innerhalb einer Box liegen die inneren 50% der Messungen, die H-Linie enthält die inneren 80% der Messungen.

In Abbildung 1 wird deutlich, daß Java (Version 1.2.1) im Schnitt vier bis fünf mal soviel Hauptspeicher verbraucht wie C/C++ und daß Java 1.2.1 den Speicher in großen Inkrementen alloziert. Die Laufzeitübersetzung (Hotspot) braucht extra Raum.
In Abbildung 2 wird deutlich, daß die individuellen Unterschiede zwischen den Experimentteilnehmern gewaltig waren, und zwar bei allen Sprachen. Die beobachteten Laufzeiten variierten von Sekunden bis zu 30 oder gar 40 Minuten.
Die Messungen zeigen ferner, daß Hotspot – das ist der neue ausführungsbegleitende Übersetzer, der manche Code-Passagen auch mehrfach während der Programmlaufzeit optimiert – für lange laufende Programme zu besseren Ergebnissen führt als der gewöhnliche Just-In-Time-Übersetzer. Ferner gibt es keinen Unterschied im Median zwischen Java und C++. Die Variabilität innerhalb einer Sprache ist deutlich höher als der Unterschied zwischen den Sprachen. Die schnellsten fünf Java-Programme sind fünf mal schneller als der Median der C++-Programme; allerdings sind diese Programme auch drei mal langsamer als die fünf schnellsten C++-Programme. Im Experiment sind die C-Programme deutlich schneller, was allerdings daran liegen kann,

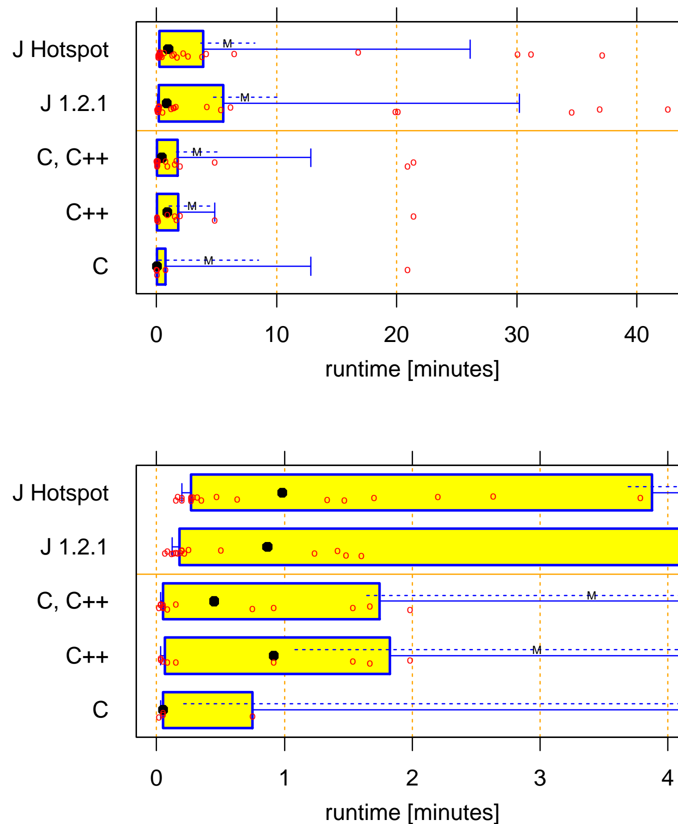

Abbildung2. Auf Solaris 7 gemessene Laufzeiten der unterschiedlichen Programme; unten vergrößert dargestellt. Legende siehe Abbildung 1.

daß nur sehr wenige Datenpunkte vorliegen und daß vermutlich die besten Experimentteilnehmer C zur Realisierung ausgewählt haben.
Zusammengefaßt läßt sich sagen, daß die Programmiererunterschiede größer als die Sprachunterschiede sind, daß C++ kaum einen Laufzeitvorteil mehr gegenüber Java hat, aber daß C++ noch immer weniger Hauptspeicher erfordert als Java.

2. **Auf wissenschaftlichem Benchmark etwa 35% langsamer als C.** Pozo und Miller haben mit SciMark 2.0 [33] einen Java-Benchmark für wissenschaftliche Anwendungen zusammengestellt, der aus den folgenden fünf mittelgroßen numerischen Kernroutinen besteht.
   - Komplexwertige schnelle Fouriertransformation
   - Gauss-Seidel-Relaxation
   - Monte Carlo Integration von e^{-x^2}

  - Multiplikation dünnbesetzter Matrizen
  - LU-Faktorisierung dichter Matrizen mir Pivotisierung

   Für jede dieser Kernroutinen stehen sowohl eine Java- als auch eine C-Version zur Verfügung. Ferner gibt es jeweils sowohl eine Version, die vollständig im Cache Platz hat als auch eine, die nicht mit dem Cache des Rechners auskommt.

   Die Ergebnisse werden auf der Web-Seite bekanntgegeben. Zum Beispiel werden von Java 1.1.8 auf einem Intel Celeron 366 unter OS/2 insgesamt etwa 76 MFlop erreicht. Zum Vergleich: die im Netscape Browser enthaltene JVM (1.1.5) erreicht mit dem gleichen Prozessor auf einer Linux-Installtion nur etwa 1% dieser Leistung.

   Laut Aussagen von Pozo erreichen die C-Versionen, deren Ergebnisse (noch) nicht auf der Web-Seite zur Verfügung stehen, im Schnitt derzeit nur noch 35% höhere Mflop-Raten (im Vergleich zu astronomischen 13500% bei einer recht schlechten früheren Java-Plattform.)

3. **Lerneffekte beschleunigen Java scheinbar.**

   Immer häufiger finden sich in Zeitschriften und auf Konferenzen Beiträge, die beschreiben, wie eine Java-Applikation „flottgemacht“ worden ist, bzw. welche Regeln befolgt werden sollten, damit von Anfang an eine schnelle Java-Applikation entsteht. Exemplarisch sei hier auf die folgenden Beiträge verwiesen: [27, 6, 35].

   Ähnlich wie es Programmierer erst erlernen mußten, wie man Programme schreibt, die von einem Übersetzer für den Einsatz auf einem Vektorrechner optimiert werden können, so müssen Programmierer wohl erst lernen, wie mit den Sprach- und Bibliothekselementen von Java sorgsam umzugehen ist, damit eine vernünftige Leistung resultiert. Insbesondere diejenigen Programmierer scheinen von einem allzu verschwenderischen Umgang mit Java-Ressourcen betroffen zu sein, die nicht auf umfangreiche Vorerfahrungen mit anderen objektorientierten Programmiersprachen und/oder Thread-Bibliotheken zurückgreifen können.

Insgesamt ist die Geschwindigkeit von Java besser als ihr Ruf. Es ist also durchaus nicht abwegig, Java ernsthaft auch als Sprache für zentrale Komponenten von Grande-Anwendungen in Betracht zu ziehen. Die Ergebnisse der beiden Arbeitsgruppen des JavaGrande-Forums, die im folgenden diskutiert werden, unterstreichen dies noch.

3 Arbeitsgruppe Numerisches Rechnen

3.1 Ziele der Arbeitsgruppe

Die Arbeitsgruppe Numerisches Rechnen des JavaGrande-Forums hat sich zum Ziel gesetzt, die Verwendbarkeit von Java für numerisches Rechnen zu evaluieren und darauf aufbauend konsensfähige Vorschläge zu erarbeiten und (mindestens prototypisch) umzusetzen, um Unzulänglichkeiten der Programmiersprache bzw. des Laufzeitsystems zu beseitigen.

Die folgenden konkreten Teilziele und die jeweils erreichten Resultate werden in den anschließenden Abschnitten präsentiert.

- Verbesserung der Fließkomma-Arithmetik
- Bereitstellung einer effizienten komplexwertigen Arithmetik
- Bereitstellung effizienter mehrdimensionaler Felder
- Verbesserung der mathematischen Bibliotheken

Die Arbeitsgruppe Numerisches Rechnen wird durch die IFIP Working Group 2.5 (International Federation for Information Processing) unterstützt.

3.2 Verbesserung der Fließkomma-Arithmetik

Für wissenschaftliches Rechnen in einer Programmiersprache ist es wichtig, daß auf den meisten Prozessoren akzeptable Geschwindigkeiten und Genauigkeiten erreicht werden. Für Grande-Applikationen ist es darüberhinaus von Bedeutung, daß auf *manchen* Prozessoren sehr hohe Fließkomma-Leistung erreicht werden kann. Ferner ist es *manchmal* von Bedeutung, daß numerische Ergebnisse exakt bis auf das letzte Bit auf unterschiedlichen Plattformen reproduzierbar sind. Numeriker haben seit Beginn der Fließkomma-Arithmetik gelernt, mit architekturspezifischen Rundungsfehlern umzugehen, so daß eine exakte Reproduzierbarkeit nur selten von eminenter Bedeutung ist. Nicht akzeptabel ist aber eine zu ungenaue Berechnung beispielsweise der trigonometrischen Funktionen in der Mathe-Bibliothek.

Diese Einschätzung steht im Gegensatz zu den Entwurfsprinzipien, die Javas Fließkomma-Arithmetik zugrunde liegen. Zentrales Entwurfsziel von Java war es, exakte Reproduzierbarkeit der Ergebnisse zu garantieren. Die erreichbare Fließkomma-Leistung spielte nur eine untergeordnete Rolle.

Fließkomma-Leistung. Um exakte Reproduzierbarkeit zu erreichen, verbietet Java daher konkret folgendes:

1. **Verwendungsverbot für 80 Bit „double extended format".** Prozessoren der x86-Familie, deren interne Register mit dem im IEEE Standard 754 definierten 80-Bit-Format arbeiten (double extended format), werden gezwungen, nach jedem einzelnen Rechenschritt die Zwischenergebnisse auf das von Java vorgeschriebene gröbere Zahlenformat zu runden. Selbst wenn mit dem Präzisionssteuerungsbit des Prozessors eine Rundung nach jedem Schritt im Register erzwungen wird, werden noch immer 15 Bit (statt standardmäßig 11 Bit) für die Darstellung des Exponenten verwendet. Der Exponent kann nur auf das von Java vorgeschriebene Maß gerundet werden, indem eine Speicherungs- und eine Wiederladeoperation zum Hauptspeicher durchgeführt wird. Neben dem Speicherverkehr bremst noch ein anderer Faktor die Fließkomma-Leistung von x86-Prozessoren: Das zweimalige Runden (unmittelbar nach der Operation und bei der Interaktion mit dem Hauptspeicher) führt in manchen Fällen zu Abweichungen von der Spezifikation in der

Größenordnung von 10^{-324}. Eine Korrektur dieses Fehlers ist zeitaufwendig und wird daher in den meisten Implementierungen nicht durchgeführt.
Diese Rundungs- und Korrekturerfordernisse sorgen dafür, daß Javas Fließkomma-Arithmetik auf Intel-Prozesoren 2 bis 10 mal langsamer ist, als nötig.

2. **Verwendungsverbot für „FMA – Fused Multiply Add“-Maschinenbefehl.** Prozessoren der PowerPC-Familie bieten einen Maschinenbefehl an, der zwei Fließkommazahlen multipliziert und eine weitere Zahl dazuaddiert, wobei nur genau eine Rundung auftritt. Dieser Maschinenbefehl kann sehr nützlich in den inneren Schleifen von Operationen auf dichtbesetzten Matrizen angewendet werden. Javas Sprachdefinition verbietet die Verwendung dieses Maschinenbefehls, obwohl in manchen Experimenten die Leistung um über 55% verbessert werden konnte.

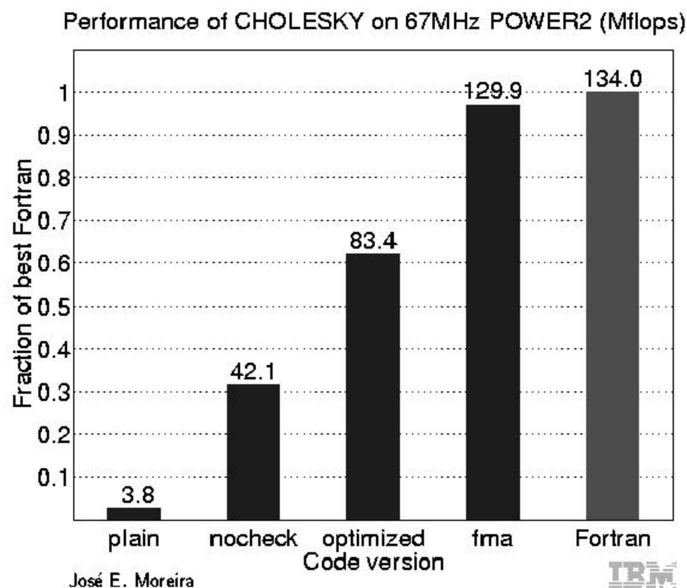

Abbildung3. IBM Experiment zu Fließkomma-Leistung und FMA

In Abbildung 3 verwendet José Moreira von IBM deren nativen Java-Übersetzer. Ohne alle Optimierung werden nur 3,8 MFlop erreicht. Wenn alle üblichen Optimierungen durchgeführt werden (einschließlich der Elimination redundanter Feldgrenzentests), dann erreicht Java 62% der Leistung eines äquivalenten Fortran-Programms (83,4 Mflop). Wenn zusätzlich der FMA-Maschinenbefehl ausgenutzt werden kann (was der Fortran-Übersetzer routinemäßig tut) können fast 97% der Fortran-Leistung erreicht werden (also knapp 55% mehr als ohne FMA).

3. **Optimierungsverbot.** Ferner sind von Java gängige Optimierungen verboten. So ist es zum Beispiel nicht erlaubt, die Assoziativität von Operatoren auszunutzen. Grund ist, daß durch diese Optimierungen evtl. eine Änderung des Rundungsverhaltens ausgelöst wird, die die exakte bitweise Reproduzierbarkeit der Ergebnisse verhindern würde.

strictfp-Schlüsselwort in Java 2. Auf Drängen des JavaGrande-Forums ist mit der Version 1.2 das Schlüsselwort `strictfp` in die Programmiersprache Java aufgenommen worden. Nur wenn dieses Attribut an Klassen oder Methoden verwendet wird, wird die bitweise Reproduzierbarkeit der Ergebnisse erzwungen. Ansonsten wird die Reproduzierbarkeit zugunsten einer Geschwindigkeitssteigerung etwas aufgeweicht. Im unattributierten Standardfall darf Java nun anonyme `double` Variablen mit 15 Bit langen Exponenten darstellen. Dadurch kann auf der x86-Familie auf die kostspieligen Speicher- und Ladeoperationen verzichtet werden.

Unterschiedliche numerische Ergebnisse ergeben sich nun nicht nur zwischen Prozessoren, mit und ohne erweiterter Exponentenlänge, sondern auch zwischen unterschiedlichen JVM-Implementierungen auf einer Plattform. Der Grund ist, daß die Verwendung der längeren Exponenten für anonyme Variablen optional ist.

Diese vom JavaGrande-Forum vorgeschlagene Semantikveränderung ersetzt den Vorschlag von Sun ersatzlos, einen Modifizierer `widefp` einzufügen, durch den das Ausmaß der Ergebnisunterschiede wesentlich dramatischer ausgefallen wäre.

Reproduzierbarkeit der `Math`-Funktionen. Neben den strikten Verboten, die die Fließkomma-Leistung beschränken, leidet Javas Fließkomma-Arithmetik auch noch unter einem Genauigkeitsproblem, das den Umgang mit Javas Fließkomma-Arithmetik erschwert.

Die ursprüngliche Spezifikation von Java legt fest, daß für die Realisierung der Bibliothek `java.lang.Math` nach Java übertragene Implementierungen aus `fdlibm` verwendet werden sollen.[2]

Offensichtlich gibt es aber keine bzw. wenige Implementierungen die sich an diese Regel halten. Stattdessen verwenden Hersteller die von der Hardware angebotenen Maschinenbefehle wodurch es zu inkorrekten Ergebnissen auf den jeweiligen Plattformen kommen kann.

Erst mit der Java 2 Plattform bietet Sun Kapselungsmethoden an, die ihrerseits auf die in C realisierten `fdlibm`-Funktionen zurückgreifen. Dadurch ist zumindest die Möglichkeit gegeben, daß zumindest die `fdlibm`-Ergebnisse auf allen Plattformen erzielt werden.

Eine ideale Mathe-Bibliothek würde Fließkomma-Zahlen liefern, die höchstens 0,5 ulp (unit in the last place) vom tatsächlichen Ergebnis entfernt

[2] Die Bibliothek `fdlibm` ist die von Sun kostenlos verbreitete Mathe-Bibliothek, die erheblich stabiler, korrekter und portabler arbeitet als die `libm`-Bibliotheken der meisten Plattformen.

liegen,[3] also so gut wie möglich gerundet worden sind. Die Bibliothek `fdlibm` selbst liefert leider auch nur eine Genauigkeit von 1 ulp, weshalb sich Sun in Java 1.3 entschlossen hat, die Spezifikation so zu ändern, daß nun die Ergebnisse der Mathe-Bibliothek einen Fehler von 1 ulp haben dürfen. Ferner sind zwei Mathe-Bibliotheken als Beitrag zur Diskussion entstanden:

- John Brophy, Visual Numerics hat eine `fdlibm` vollständig in Java implementiert, so daß in Zukunft auf die Kapselungsmethoden verzichtet werden kann[1].
- Abraham Ziv, IBM Haifa hat eine Mathe-Bibliothek (in ANSI C) erstellt, die korrekt rundet (Fehler höchstens 0,5 ulp) [41].

Vorschlag `fastfp`-Schlüsselwort. Das JavaGrande-Forum erarbeitet derzeit einen JSR (Java Specification Request),[4] der vorschlägt, einen weiteren Modifizierer `fastfp` in die Sprache einzufügen. In Klassen und Methoden die mit diesem Modifizierer attributiert sind, soll die Verwendung des FMA-Maschinenbefehls erlaubt sein. Ferner soll die Mathe-Bibliothek um eine FMA-Methode erweitert werden, deren Benutzung durch den Programmierer die Verwendung des FMA-Befehls erzwingt. Ferner wird untersucht, ob es sinnvoll ist, durch den Modifizierer auch Assoziativitätsoptimierungen zuzulassen.

3.3 Effiziente komplexwertige Arithmetik

Eine Voraussetzung für den Einsatz von Java im wissenschaftlichen Rechnen ist die effiziente und bequeme Unterstützung komplexer Zahlen.

Mit den Sprachmitteln von Java besteht die einzig sinnvolle Möglichkeit zur Verwendung komplexer Zahlen darin, eine `Complex`-Klasse zu erstellen, deren Objekte zwei `double`-Werte enthalten. Komplexwertige Arithmetik muß dann umständlich durch Methodenaufrufe ausgedrückt werden, wie in folgendem Code-Fragment.

```
Complex a = new Complex(5,2);
Complex b = a.plus(a);
```

Dies hat drei Nachteile: Ohne Operatorüberladung sind erstens arithmetische Ausdrücke nach ihrer Formulierung nur schwer lesbar. Zweitens ist wegen fehlender Sprach- und Übersetzerunterstützung sogenannter Wertklassen in Java das Anlegen eines Objekts viel langsamer und verbraucht mehr Speicherplatz als das Anlegen einer Variable eines primitiven Typs. Klassenbasierte komplexwertige Arithmetik ist dadurch erheblich langsamer als Javas primitivwertige Arithmetik. Dieser negative Effekt wird noch verstärkt, da durch die Formulierung mit

[3] Für Fließkomma-Zahlen zwischen 2^k und 2^{k+1} ist ein ulp 2^{k-52}.

[4] Der offizielle Weg, Änderungen in der Programmiersprache Java oder ihrer Laufzeitumgebung zu bewirken, ist es , einen JSR einzureichen [24]. Je nach JSR berät unter anderem ein Gutachtergremium über den Änderungsvorschlag; ferner wird die Nutzergemeinde um Stellungnahme gebeten.

Hilfe von Methodenaufrufen Hilfsobjekte angelegt werden, die bei Verwendung der Stapelmaschine zur Wertspeicherung gar nicht erforderlich wären. Drittens fügen sich klassenbasierte komplexe Zahlen grundsätzlich nicht voll in das übliche Erscheinungsbild der primitiven Typen ein: Sie sind nicht in die Typbeziehungen integriert, die zwischen Javas primitiven Typen bestehen, so daß z.B. die Zuweisung eines primitiven `double`-Wertes zu einem `Complex`-Objekt keinesfalls eine automatische Typkonvertierung auslöst, was bei primitiven komplexen Zahlen möglich wäre. Gleichheitstests zwischen komplexen Objekten beziehen sich auf Objektidentitäten statt auf Wertgleichheit. Darüberhinaus ist bei einer klassenbasierten Lösung stets ein Konstruktoraufruf erforderlich, wo ein Literal zur Repräsentation eines konstanten Werts ausreichen sollte.

Da wissenschaftliches Rechnen nur einen unbedeutenden Anteil an der gesamten Java-Nutzung hat, ist es sehr unwahrscheinlich, daß die Java Virtual Machine (JVM) bzw. der Bytecode um einen neuen primitiven Typ `complex` erweitert wird, was sicherlich das beste Vorgehen zur Einführung komplexer Zahlen in Java wäre. Da ferner nicht abzusehen ist, ob (und wenn ja, wann) Java um allgemeine Überladbarkeit von Operatoren und um effiziente Unterstützung von Wertklassen erweitert wird, und da eine nahtlose Einbindung einer Klasse in das Erscheinungsbild existierender primitiver Typen grundsätzlich auch nach einer solchen Erweiterung nicht gegeben ist, hält es das JavaGrande-Forum für sinnvoll, auf zwei verschiedenen Wegen effiziente und bequeme komplexwertige Arithmetik für Java vorzuschlagen.

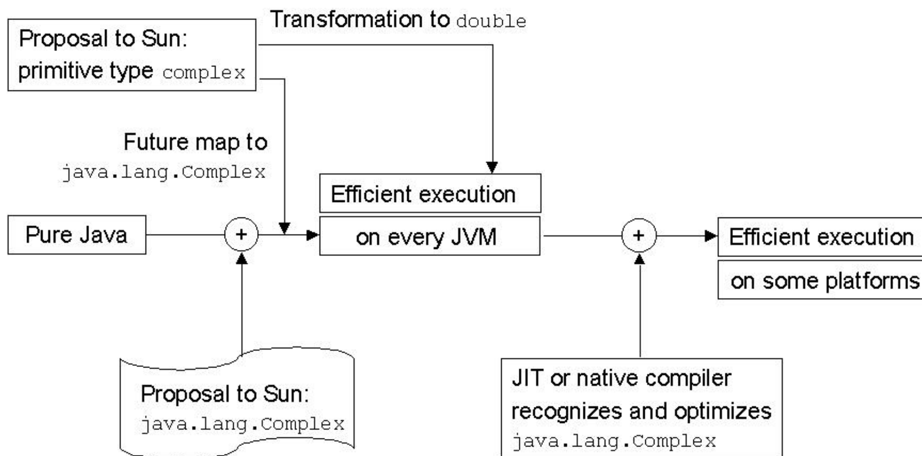

Abbildung4. Strategien zur Einführung komplexwertiger Arithmetik

Klasse `java.lang.Complex`. Abbildung 4 zeigt, daß das JavaGrande-Forum einerseits eine Bibliothek `java.lang.Complex` definieren und prototypisch realisieren wird. Diese Bibliothek lehnt sich im Stil an die anderen Zahlen-Klassen an, die Java bereits standardmäßig liefert. Zusätzlich werden allerdings arithmetische Operationen vorhanden sein, so daß es möglich ist, methodenbasierte Operationen auf komplexen Zahlen auszudrücken. Die Klasse wird Sun in Form eines JSR unterbreitet.

IBM hat die Leistung einer klassenbasierten komplexen Arithmetik untersucht. Bei einer Jacobi-Relaxation erreicht eine komplexwertige Implementierung nur etwa 2% der Geschwindigkeit der entsprechenden auf `double` beruhenden Implementierung. Der Speicherverbrauch ist in Abbildung 5 dargestellt.

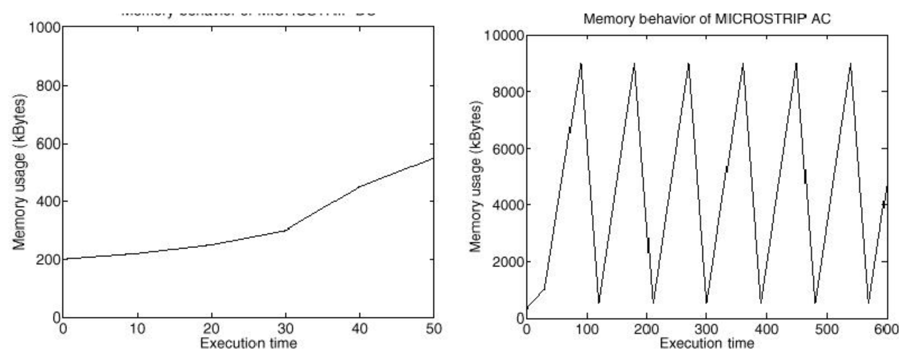

Abbildung5. Speicherverbrauch einer objektbasierten komplexwertigen Jacobi-Relaxation. Es wird deutlich, daß nach einem gemächlichen Start große Mengen an temporären Objekten erzeugt werden, die dann jeweils vom Speicherbereiniger wieder freigegeben werden müssen.

Aufgrund dieser schlechten Ergebnisse hat IBM die Semantik der `Complex`-Klasse fest in ihrem nativen Java-Übersetzer eingebaut, intern wird ein komplexer Datentyp verwendet und darauf werden die zugehörigen Optimierungen durchgeführt [40]. Die in Java-Notation vorhandenen Methodenaufrufe und Objekterzeugungen werden weitestgehend entfernt bzw. über den Stapel abgewickelt, so daß man nun von einer effizienten Unterstützung der Klasse `Complex` sprechen kann.

Primitiver Datentyp `complex`. Als weiteren JSR schlägt das JavaGrande-Forum vor, einen primitiven Datentyp `complex` mit zugehörigen Infix-Operationen in die Sprache einzuführen, diesen aber in einem Präprozessorschritt auf übliches Java zurückzuführen. Dadurch brauchen weder das ByteCode-Format noch existierende JVM-Implementierung verändert zu werden.

Wie in Abbildung 4 gezeigt werden zwei Transformationen vorgesehen. Erstens wird der primitive Datentyp `complex` auf ein Paar von `double`-Werten abgebildet. Zweitens besteht die Möglichkeit, `complex` auf die `Complex`-Klasse

abzubilden, damit auf den Plattformen, auf denen sie zur Verfügung steht, die Übersetzerunterstützung ausgenutzt werden kann.

Mit dem Übersetzer *cj* ist an der Universität Karlsruhe die Umsetzung des primitiven Datentyps `complex` sowohl formal beschrieben als auch prototypisch relaisiert worden [16, 15]. Diese Arbeiten bilden die Grundlagen des JSR, wobei derzeit diskutiert wird, ob neben dem primitiven Datentyp `complex` auch ein weiterer primitiver Datentyp `imaginary` eingeführt werden soll, wie dies auch für C9X evaluiert wird [26, 2].

3.4 Effiziente mehrdimensionale Felder

Genauso, wie komplexwertige Arithmetik effizient und bequem verfügbar sein muß, ist numerisches Rechnen ohne effiziente (d.h. optimierbare) und bequeme mehrdimensionale Felder undenkbar.

Java bietet mehrdimensionale Felder als Felder von eindimensionalen Feldern an. Die Probleme bestehen darin, daß erstens diverse „Zeilen“ durch ein gemeinsames eindimensionales Feld realisiert werden könnten (Alias), zweitens die „Zeilen“ unterschiedliche Länge haben könnten, jeder Zugriff auf mehrdimensionale Felder mindestens eine Indirektionsstufe zur Zeigerverfolgung erzwingt und schließlich jeder Feldzugriff zur Laufzeit darauf überprüft wird, ob er innerhalb der Feldgrenzen liegt.

Obwohl in vielen Fällen durch Datenflußanalyse festgestellt werden kann, daß Feldzugriffe innerhalb der Feldgrenzen liegen werden, oder obwohl in vielen Fällen der Code so vervielfacht und mit Randbedingungen versehen werden kann, daß in manchen Passagen außer den Randtests keine individuellen Tests der Feldzugriffe erfolgen müssen, sind Javas Felder dennoch schwer zu optimieren, weil Aliase und unterschiedliche „Zeilen“-Längen eine Abbildung der Indexvektoren auf Speicheradressen verhindern.

Aus diesem Grunde schlägt das JavaGrande-Forum (wieder in Form eines JSR) eine Bibliothek für mehrdimensionale Felder vor, die mit einer festgelegten Ausrollreihenfolge, auf eindimensionale Java-Felder abgebildet werden. Durch die festgelegte Ausrollreihenfolge der Dimensionen wird es möglich, Algorithmen zu verwenden, die durch Kenntnis der relativen Speicheranordnung besseres Cache-Verhalten zeigen.

Wie zuvor bei den komplexen Zahlen, wird die Verwendung der Klassen im Komfort zu Wünschen übrig lassen, weil statt der eleganten `[]`-Notation nun mit Zugriffsmethoden gearbeitet werden muß.

Daher bietet sich auch hier ein zweigleisiges Vorgehen an. Einerseits wird IBM Unterstützung für Klassen der mehrdimensionalen Felder fest in ihren nativen Übersetzer einbauen. Gleichzeitig wird eine erweiterte Feldzugriffssyntax erarbeitet, mit der elegant auf mehrdimensionale Felder zugegriffen werden kann. Diese Syntaxerweiterung, die wegen der benötigten Interaktion mit regulären eindimensionalen Feldern von Java recht diffizil ist, und der zugehörige Präprozessor werden Sun ebenfalls als JSR vorgeschlagen.

3.5 Grundsätzlichere Überlegungen

Die allgemeine Lösung sowohl für komplexe Zahlen als auch für mehrdimensionale Felder besteht darin, leichtgewichtige Klassen und Operator-Überladung in Java einzuführen.

Leichtgewichtige Klassen haben Wert-Semantik – ihre Instanzvariablen können also nach der Objekterzeugung nicht verändert werden. Daduch entfallen die Probleme der Gleichheitssemantik. Ferner können leichtgewichtige Objekte in vielen Fällen auch auf dem Stapel alloziert und über den Methodenaufrufstapel per Kopie übergeben werden, wodurch sich Leistungsverbesserungen erzielen lassen. Wenn es dem Programmierer ermöglicht wird, die Basisoperationen `+`, `-`, `*`, `/` und `[]` zu überschreiben und selbst zu implementieren, werden die Präprozessor-Lösungen hinfällig. Ferner wird es möglich, auch z.B. eine Intervallarithmetik mit den Bordmitteln der Sprache zu realisieren.

Warum versucht das JavaGrande-Forum also nicht, leichtgewichtige Klassen und Operator-Überladung in Java einzuführen? Die Antwort ist pragmatisch: Das JavaGrande-Forum hofft, daß die oben aufgeführten JSR leicht genug sind, um den Prozeß der Sprachänderung zu überstehen. Die Gemeinde der Java-Benutzer ist praktisch nicht betroffen und bemerkt die Änderungen vermutlich kaum; die wenigsten der heutigen Java-Nutzer sind über die Existenz, geschweige denn die Bedeutung, des `strictfp`-Schlüsselworts im Bilde. Je kleiner der Anteil der Betroffenen ist, desto unbeschadeter wird ein JSR befürwortet.

Wertklassen und Operator-Überladung erfordern eine erhebliche Veränderung der Sprache als Ganzes und betreffen (vermutlich) auch den ByteCode und damit die JVM. Derartige Änderungsvorschläge werden ganz erhebliche Diskussionen hervorrufen. Insbesondere wegen des schon fast religiösen Charakters des Operator-Überladens ist der Ausgang derartiger Bemühungen als offen zu betrachten, während die übrigen Vorschläge recht chancenreich sind.

Dennoch erarbeitet das JavaGrande-Forum zusätzlich zu den oben genannten Vorschlägen, auch Beiträge zu diesen grundsätzlicheren Themen.

4 Arbeitsgruppe Parallelismus und Verteilung

4.1 Ziele der Arbeitsgruppe

Die Arbeitsgruppe Parallelismus und Verteilung des JavaGrande-Forums hat sich zum Ziel gesetzt, die Verwendbarkeit von Java für paralleles und verteiltes Rechnen zu evaluieren und darauf aufbauend konsensfähige Aktionen zu konzipieren und zu ergreifen, um Unzulänglichkeiten der Programmiersprache bzw. des Laufzeitsystems zu beseitigen.

Die folgenden konkreten Teilziele und die jeweils erreichten Resultate werden in den anschließenden Abschnitten präsentiert.

- Beschleunigung des entfernten Methodenaufrufs
- Bereitstellung von „Message Passing“
- Benchmarks

4.2 Schneller entfernter Methodenaufruf

Soll in Java verteilt parallel programmiert werden, dann ist es essentiell, daß Javas Bibliotheksmittel gute Latenzzeiten und eine hohe Bandbreite zulassen.

Der entfernte Methodenaufruf gängiger Java-Implementierungen (RMI) ist aber zu langsam für Hochleistungsanwendungen, da RMI für Weitverkehrskommunikation entworfen wurde, auf einer langsamen Objektserialisierung aufbaut und keine Hochgeschwindigkeitsnetze unterstützt. Je nach Typ und Anzahl der mit einem entfernten Methodenaufruf versendeten Argumente dauert dieser mit Java im Millisekundenbereich, wobei etwa ein Drittel der Zeit auf die RMI-Implementierung, ein Drittel auf die Serialisierung der Argumente (also deren Überführung in eine maschinenunabhängige Byte-Repräsentation) und ein Drittel auf Kosten der Übertragung (bei TCP/IP-Ethernet) gehen.

Um einen wirklich schnellen entfernten Methodenaufruf zu erreichen, muß auf allen Ebenen gearbeitet werden, d.h. man benötigt eine schnelle Serialisierung, eine schnelle RMI-Implementierung und die Möglichkeit, auch Kommunikationshardware zu verwenden, die nicht notwendigerweise das TCP/IP-Protokoll benutzt.

An der Universität Karlsruhe wurden im Rahmen des JavaParty-Projekts [23] alle drei Aufgaben angegangen und die derzeit weltweit schnellste Java-Implementierung eines entfernten Methodenaufrufs gebaut, der auch über Myrinet-Kommunikationskarten abgewickelt werden kann.

Die schnelle UKA-Serialisierung [17] kann statt der offiziellen Serialisierung (und als Ergänzung dieser) verwendet werden und spart 76%–96% der Serialisierungzeit ein. Die zentralen Ideen der schnellen Serialisierung sind:

- Explizite Serialisierungsroutinen („Marshalling-Routinen“) sind schneller als das vom klassischen RMI verwendete autmatische Überführen in eine Byte-Repräsentation mit Hilfe von Typ-Introspektion.
- Einen wesentlichen Anteil an den Kosten der Serialisierung hat die aufwendige Verschlüsselung der Typinformation, die es ermöglicht, serialisierte Java-Objekte auch mit späteren Versionen des Programms wieder zu laden. Für Kommunikationszwecke, insbesondere in Rechnerbündeln, die über ein gemeinsames Dateisystem verfügen, ist eine schlankere Typverschlüsselung schneller.
  Das JavaGrande-Forum hat Sun Microsystems davon überzeugt, daß es sinnvoll ist, die Art der Typverschlüsselung wählbar zu machen. Die Serialisierung wird in einer der nächsten Versionen diese Option anbieten.
- Da bei RMI Objekte kopiert werden, ist es grundsätzlich aus Gründen des Semantik erforderlich, diese bei jedem Aufruf erneut zu übertragen. Leider unterscheidet RMI nicht zwischen Typverschlüsselung und Nutzdaten, so daß auch die aufwendigen Typinformationen bei jedem Methodenaufruf erneut übertragen werden.
  Sun hat angekündigt, auch dieses Idee des getrennten Zurücksetzens von Typinformation und Nutzdaten aufzugreifen; jedoch ist bis jetzt keine konkrete Version genannt.

- Die offizielle Serialisierung nutzt mehrere Schichten von Strömen, die alle über eigene Pufferung verfügen. Für schnelle Kommunikation, die idealerweise ohne irgendwelche Kopieroperationen auskommt, führt diese Schichtung mit der Vielzahl der benötigten Kopieroperationen zu inakzeptabler Leistung, insbesondere, weil die Puffer auch noch unterschiedliche Größen haben.
  Die UKA-Serialisierung kommt mit einem Puffer aus, in den die Byte-Repräsentation unmittelbar geschrieben wird.
  Sun wird an der Schichtung aus Gründen eines klareren objektorientierten Entwurfs festhalten, jedoch wird zumindest die interne Implementierung der Schichten verbessert werden.
- Die JVM bietet keine Möglichkeit an, die Byte-Repräsentation eines `float`- oder `double`-Wertes zu erfragen, ohne über das native Interface (JNI) eine C-Routine zu verwenden. Leider verwendet die offizielle Serialisierung den recht langsamen JNI-Übergang für jedes einzelne Element von Feldern dieser primitiven Typen.
  Die UKA-Serialisierung verwendet (optional) eine eigene JNI-Routine, die die Byte-Repräsentation eines Feldes von Fließkomma-Werten als Ganzes zurückliefert, sodaß der langsame JNI-Übergang nur einmal strapaziert zu werden braucht.
  Die Hotspot-Arbeitsgruppe bei Sun hat angekündigt, diese Ergebnisse aufzugreifen.

In Karlsruhe ist neben der UKA-Serialisierung auch eine Ersatz-Implementierung des RMI entstanden. KaRMI [29, 30] kann statt des offiziellen RMI verwendet werden und beseitigt unter anderem die folgenden Unzulänglichkeiten im offiziellen RMI:

- KaRMI ermöglicht die Verwendung von Nicht-TCP/IP-Netzwerken. Aufgrund der Arbeiten des JavaGrande-Forums plant Sun langfristig, auch in der offiziellen RMI-Version die Unterstützung anderer Netze möglich zu machen, jedoch sind die Vorstellungen von Sun derzeit noch schemenhaft.
- Obwohl die RMI-Dokumentation von einer klaren Schichtung spricht, ist diese in der zugehörigen Implementierung bestenfalls vage zu erkennen. KaRMI hat eine klare Schichtung, durch dies es möglich wird, andere Protokollsemantiken (z.B. Multicast) und andere Kommunikationshardware (z.B. Myrinet-Karten) anzusprechen.
- RMI bietet dem Programmierer die Möglichkeit, Objekte an feste Port-Nummern zu exportieren, wodurch ein Detail der Netzwerkschicht bis zum Anwender durchgereicht wird. Dies widerspricht dem Prinzip, Details der unteren Schichten vor höheren Schichten zu verbergen.
  KaRMI hingegen unterstützt diese Funktionalität nur, wenn das unterliegende Kommunikationsnetzwerk Portnummern anbietet.
- Im Gegensatz zu KaRMI ist die Implementierung des offiziellen RMI nicht für Geschwindigkeit optimiert. Ausgelöst von den guten Ergebnissen von KaRMI hat Sun Personal eingestellt, um die RMI-Implementierung zu optimieren.

– Der verteilte Speicherbereiniger des offiziellen RMI ist für Weitverkehrsnetze ausgelegt. Obwohl es für enggekoppelte Rechnerbündel (und andere Plattformen) optimierte Speicherbereiniger gibt [31], sieht das offizielle RMI im Gegensatz zu KaRMI keine Möglichkeit vor, einen alternativen Speicherbereiniger zu nutzen.

Auf einem Bündel von DEC Alpha Rechnern, die mit Myrinet-Karten verbunden sind, kann ein entfernter Methodenaufruf derzeit vollständig[5] in Java in etwa 80 μs ausgeführt werden. Abbildung 6 zeigt, daß bei Benchmark-Programmen bis zu 96% der Laufzeit eingespart werden können, wenn die UKA-Serialisierung, das schnelle RMI und die schnellere Kommunikationshardware eingesetzt wird.

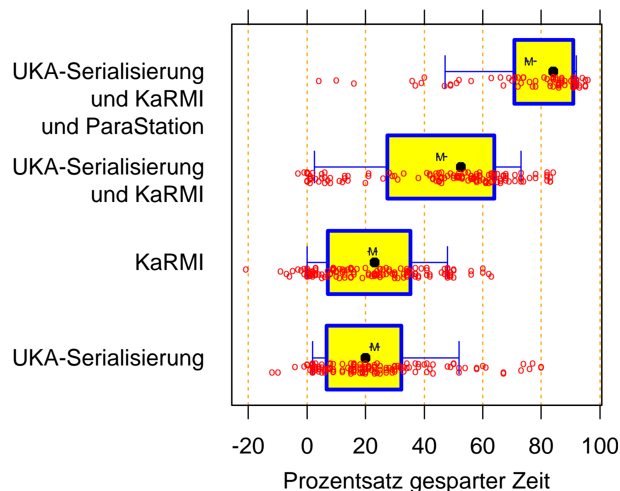

Abbildung6. Die unteren drei „Zeilen“ zeigen je $2 \cdot 64$ Meßergebnisse für diverse Benchmarks (Ethernet auf PC und FastEthernet auf Alpha). Die unterste Zeile zeigt die erzielte Laufzeitverbesserung von RMI mit UKA-Serialisierung. Die zweite Zeile zeigt die Verbesserung, die KaRMI zusammen mit der JDK-Serialisierung bewirkt. Die darüberliegende Zeile zeigt die kombinierte Wirkung. Die oberste Zeile demonstriert das Verhalten, wenn zusätzlich zur UKA-Serialisierung und KaRMI noch die Myrinet-Kommunikationskarten verwendet werden (64 Meßergebnisse).

4.3 Message Passing in Java

Während Javas Mechanismen für paralleles und verteiltes Programmieren am Client/Server-Modell orientiert sind, hat sich mit MPI bereits länger ein symmetrisches, auf Botschaftenaustausch basierendes, paralleles Berechnungsmodell etabliert.

[5] Die Anbindung des Natzwerktreibers selbst ist nicht in Java realisiert.

Um die Erfahrung aus MPI-basierten Lösungen beim Übergang auf Java nicht zu verlieren, arbeitet eine Untergruppe des JavaGrande-Forums daran, eine MPI-Bindung für Java bereitzustellen. Durch die Verfügbarkeit einer solchen Schnittstelle würden MPI-basierte Grande-Anwendungen möglich.

Mitglieder des JavaGrande-Forums haben daher einige Vorschläge hervorgebracht und prototypisch realisiert.

- mpiJava: eine Sammlung von Kapselklassen, die über die native Schnittstelle von Java (JNI) auf die C++-Bindung von MPI zurückgreifen [4, 18].
- JavaMPI: automatisch für ein Programm erzeugte JNI-Kapselungen für die C-Bindung von MPI [28].
- MPIJ: eine Implementierung der MPI-Schnittstelle in Java, die sich stark an der C++-Bindung orientiert und recht gute Leistungsergebnisse vorzuweisen hat [25].

Derzeit arbeitet die Untergruppe an einer Vereinheitlichung der bisherigen Prototypen [3]. Als wesentliche Frage stellt sich allerdings, wie die Mechanismen von Java in Zukunft im Zusammenhang mit MPI verwendbar gemacht werden können. Untersucht wird derzeit, ob die in MPI verwendbaren Typen nicht um Java-Objekte erweitert werden könnten, die dann in serialisierter Form verschickt würden [5]. Hierbei spielt die schnelle Serialisierung (s.o.) eine wichtige Rolle. Ferner wird geprüft, ob und wie Javas Thread-Modell verwendet werden kann, um die prozeßbasierten Ansätze, die MPI zugrundeliegen, zu erweitern.

4.4 Benchmarks

Das JavaGrande-Forum hat eine Benchmark-Initiative gestartet, deren Anspruch es ist, für Grande-Applikationen aussagekräftige Angaben zu machen, bzw. Schwächen der Java-Plattform aufzudecken.

Die Federführung dieser Initiative liegt bei EPCC (Edinburgh) [7]. Derzeit existiert eine stabile Sammlung von nicht-parallelen Benchmarks in drei Kategorien:

- Basisoperationen
  - Arith: Ausführung arithmetischer Operationen
  - Assign: Zuweisung zu Variablen
  - Cast: Typkonvertierung
  - Create: Erzeugung von Objekten und Feldern
  - Loop: Schleifenoverhead
  - Math: Ausführung von Methoden der Mathe-Bibliothek
  - Method: Methodenaufruf
  - Serial: Serialisierung
  - Exception: Behandlung von Exceptions
- Berechnungskerne
  - Series: Fourier-Koeffizientenanalyse
  - LUFact: LU Faktorisierung
  - SOR: Gauss-Seidel-Relaxation

  - HeapSort: Sortierung von Integer-Werten
  - Crypt: IDEA-Verschlüsselung
  - FFT: Schnelle Fouriertransformation
  - Sparse: Multiplikation dünnbesetzer Matrizen
- Applikationen
  - Search: Alpha-beta-Suche mit Abschneiden
  - Euler: Computational Fluid Dynamics
  - MD: Simulation von Moleküldynamik
  - MC: Monte-Carlo-Simulation
  - Ray Tracer: 3D-Strahlverfolgung

Derzeit wird an Thread-Benchmarks für alle drei Kategorien gearbeitet. Es werden dabei die Basisoperationen (create, join, barrier, synchronized methods) vermessen; für die meisten Berechnungskerne liegen parallelisierte Versionen vor; einige der Applikationen (Monte Carlo und Raytracer) werden ebenfalls parallel implementiert.

Ferner wird daran gearbeitet, eine möglichst vollständige Menge von äquivalenten Implementierungen in C/C++ zu erstellen, damit ein fundierter Sprachvergleich möglich wird.

Im nächsten Schritt werden dedizierte RMI- und MPI-Benchmarks ergänzt, z.B. [19, 32].

5 Zusammenfassung

Die wesentlichen bisherigen Beiträge des JavaGrande-Forums sind die Schlüsselworte `strictfp` und `fastfp` für eine verbesserte Fließkomma-Arithmetik, die Arbeiten auf dem Gebiet der komplexen Zahlen und der mehrdimensionalen Felder, die schnelle Serialisierung und das schnelle RMI und schließlich die Benchmark-Initiative.

Durch die Zusammenarbeit mit Sun Microsystems, durch die Eröffnung eines neuen Forschungszweigs und durch die Bündelung der Interessen der „JavaGrande-Gemeinde" ist zu hoffen, daß auch in Zukunft die Anforderungen des Hochleistungsrechnens Eingang in die Fortentwicklung der Programmiersprache Java und ihrer Laufzeitumgebung finden werden.

Danksagungen

Diese Zusammenstellung der Aktivitäten und Ergebnisse des JavaGrande-Forums verarbeitet gesammelte Präentationsunterlagen von Geoffrey Fox, Dennis Gannon, Roldan Pozo. Ein Dankeschön auch an Sun Microsystems, insbesondere an Sia Zadeh, für die inhaltliche und finanzielle Unterstützung der Arbeiten des JavaGrande-Forums.

Literatur

1. John Brophy. Prototype implementation of `java.lang.complex`. http://www.vni.com/corner/garage/grande/.
2. C9x proposal. ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/ and http://anubis.dkuug.dk/jtc1/sc22/wg14/.
3. B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-003, Java Gande Forum, November 1998.
4. Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with "HPJava". *Concurrency: Practice and Experience*, 9(6):579–619, June 1997.
5. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface for MPI. In *ACM 1999 Java Grande Conference*, pages 66–71, San Francisco, 1999.
6. Bruce Eckel. *Thinking in Java.* Prentice Hall, Englewood Cliffs, New Jersey, 1998.
7. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/.
8. J. Rolim et al., editor. *Parallel and Distributed Processing.* Number 1586 in Lecture Notes in Computer Science. Springer Verlag, Puerto Rico, April 12, 1999.
9. *Proc. First UK Workshop on Java for High Performance Network Computing at EuroPar'98.* Southampton, September 2–3, 1998.
10. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(6). John Wiley & Sons, June 1997.
11. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(11). John Wiley & Sons, November 1997.
12. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 10(11–13). John Wiley & Sons, September–November 1998.
13. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume to appear. John Wiley & Sons, 2000.
14. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, Reading, Mass., 1996.
15. Edwin Günthner and Michael Philippsen. Complex numbers for Java. In *Proc. ISCOPE'99, 3rd International Symposium on Computing in Object-Oriented Parallel Environments*, number 1732 in Lecture Notes in Computer Science, pages 1–12, San Francisco, December 7–10, 1999. Springer-Verlag Berlin, Heidelberg, New York.
16. Edwin Günthner and Michael Philippsen. Komplexe Zahlen für Java. In *JIT'99, Java-Informations-Tage*, pages 253–266, Düsseldorf, September 20–21, 1999.
17. Bernhard Haumacher and Michael Philippsen. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in Lecture Notes in Computer Science, Puerto Rico, April 12, 1999. Springer Verlag.
18. http://www.npac.syr.edu/projects/pcrc/HPJava/.
19. Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1154, September–November 1998.
20. Java Grande Forum. http://www.javagrande.org.
21. Java Grande Forum, mailinglist. All Members: javagrandeforum@npac.syr.edu, Subscribe: gcf@syracuse.edu.
22. Java Grande Forum, Europe. http://www.irisa.fr/EuroTools/Sigs/Java.html.
23. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
24. The Java community process manual. http://java.sun.com/aboutJava/community process/java_community_process.html.
25. Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of MPI in Java. In *ACM 1999 Java Grande Conference*, pages 58–65, San Francisco, 1999.
26. William Kahan and J. W. Thomas. Augmenting a programming language with complex arithmetics. Technical Report No. 91/667, University of California at Berkeley, Department of Computer Science, December 1991.
27. Reinhard Klemm. Practical guideline for boosting Java server performance. In *ACM 1999 Java Grande Conference*, pages 25–34, San Francisco, 1999.

28. S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Recent Advances in PVM and MPI*, Lecture Notes in Computer Science, pages 135–142. Springer-Verlag Berlin, Heidelberg, New York, November 1997.
29. Christian Nester, Michael Philippsen, and Bernhard Haumacher. Ein effizientes RMI für Java. In *JIT'99, Java-Informations-Tage*, pages 135–148, Düsseldorf, September 20–21, 1999.
30. Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI. In *ACM 1999 Java Grande Conference*, pages 152–159, San Francisco, 1999.
31. Michael Philippsen. Cooperating distributed garbage collectors for clusters and beyond. In *Proc. 8th Int. Workshop on Compilers for Parallel Computers CPC'2000*, page to appear, Aussois, France, January 4–7 2000.
32. Michael Philippsen, Matthias Jacob, and Martin Karrenbach. Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java. *Informatik—Forschung und Entwicklung*, 13(2):72–78, 1998.
33. Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark/.
34. Lutz Prechelt. Comparing Java vs. C/C++ efficiency differences to inter-personal differences. *Communications of the ACM*, 42(10):109–112, October 1999.
35. Mark Roulo. Accelerate your Java apps! *Java World*, September 1998.
36. R. R. Oldehoeft S. Matsuoka and M. Tholburn, editors. *Proc. ISCOPE'99, 3rd International Symposium on Computing in Object-Oriented Parallel Environments.* Number 1732 in Lecture Notes in Computer Science. Springer-Verlag Berlin, Heidelberg, New York, San Francisco, December 7–10, 1999.
37. P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors. *Proc. 7th Intl. Conf. on High Performance Computing and Networking, HPCN Europe 1999.* Number 1593 in Lecture Notes in Computer Science. Springer Verlag, Amsterdam, April 12–14, 1999.
38. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, November 7–13, 1998. panel handout.
39. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Iterim Java Grande Forum Report. In *ACM Java Grande Conference'99*, San Francisco, June 14–17, 1999.
40. Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, pages 109–118, San Francisco, 1999.
41. Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, (17):410–423, 1991. www.alphaWorks.ibm.com/tech/mathlibraries4java/.