Informatik-Spektrum Manuscript-Nr.
(will be inserted by hand later)

JavaGrande – Hochleistungsrechnen mit Java

Michael Philippsen

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5, 76128 Karlsruhe

31. Januar 2000

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 ihrer Hilfe effizient zu realisieren.

Beiträge des JavaGrande-Forums sind die Schlüsselworte `strictfp` und `fastfp` für eine verbesserte Fließkomma-Arithmetik, Arbeiten zu komplexen Zahlen und zu mehrdimensionalen Feldern, sowie eine schnelle Serialisierung und ein schnelle RMI. Das JavaGrande-Forum hat einen Forschungszweig eröffnet, dessen Ergebnisse Eingang in die Fortentwicklung der Programmiersprache Java und ihrer Laufzeitumgebung fanden und finden. Es wird international und auch bei Sun Microsystems beachtet.

Schlüsselwörter: JavaGrande, strictfp, FMA, komplexe Zahlen, entfernter Methodenaufruf

Summary. The JavaGrande-Forum is a group of users, researchers, and people from industry. They either try to use Java for resource-intensive applications or try to improve the Java platform so that big applications can be done efficiently in Java.

The forum has contributed the keywords `strictfp` and `fastfp` for an improved floating point arithmetic. It has worked on complex numbers, multidimensional array, fast object serialization, and a fast RMI. Results from the new field of research that has been started by the JavaGrande-Forum are recognized internationally and within Sun Microsystems.

Key words: JavaGrande, strictfp, FMA, complex numbers, remote method invocation

Computing Reviews Classification: ???

1. JavaGrande-Forum

Inspiriert durch den Kaffeehausjargon, etablierte sich in letzter Zeit das Schlagwort der *Grande*-Anwendungen.[1] Eine Grande-Anwendung ist im wissenschaftlichen, ingenieurmäßigen oder kommerziellen Rechnen angesiedelt und zeichnet sich durch komplexe rechen-, daten- oder ein-/ausgabeintensive Bearbeitungsschritte aus. Typische Anwendungsklassen sind Modellbildung, Simulation oder Datenauswertung. Zur Bewältigung der Bearbeitungsschritte erfordert eine Grande-Anwendung sehr hohe Rechenleistung, eventuell sogar Parallelität oder verteiltes Rechnen.

Das *JavaGrande-Forum* [17] ist ein Zusammenschluß derjenigen Anwender, Forscher und Firmenvertreter, die entweder versuchen, Grande-Anwendungen mit Hilfe der Programmiersprache Java 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.

Das JavaGrande-Forum wurde im März 1998 in einer „Birds-of-a-Feather“-Sitzung in Palo Alto gegründet. Seitdem veranstaltet es regelmäßige Treffen, die ebenso wie die Web-Seite [17] und die Mailingliste [18] offen für alle Interessenten sind. Als wissenschaftlicher Koordinator fungiert Geoffrey C. Fox (Syracuse); zentraler Kontaktmann bei Sun Microsystems ist Sia Zadeh.

1.1. 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 Standardisierungskommitee)

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

3. Erarbeitung von konsensfähigen Prototypimplementierungen, Schnittstellen (APIs) und Änderungsvorschlägen, um Java und ihre Laufzeitumgebung für Grande-Applikationen verwendbar zu machen.

1.2. Mitglieder des JavaGrande-Forums

Am JavaGrande-Forum beteiligen sich vorwiegend amerikanische Firmen, Forschungseinrichtungen und Labors. In letzter Zeit sind auch in Europa JavaGrande-Aktivitäten zu beobachten [19].

Neben Sun Microsystems beteiligen sich unter anderen IBM und Intel sowie Least Square, MathWorks, NAG, MPI Software Technologies und Visual Numerics. 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, 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 widmet sich der Arbeitsgruppe Parallelismus und Verteilung.

Bisherige 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 Zuhörern der JavaOne 1999 die herausragenden Arbeiten des JavaGrande-Forums lobte.

1.3. Wissenschaftliche Arbeit des JavaGrande-Forums

Das Forum veranstaltet wissenschaftliche Konferenzen und Workshops oder ist auf Panels vertreten; siehe Tabelle 1. Die zentrale jährliche Veranstaltung des Forums ist die ACM Java Grande Konferenz. Ein Großteil der wissenschaftlichen Arbeiten der „JavaGrande-Gemeinde“ ist in den Tagungsbänden (Tabelle 1) und in mehreren Ausgaben der Zeitschrift *Concurrency – Practice & Experience* dokumentiert [9, 10, 11, 12]. Ferner gibt das JavaGrande-Forum in regelmäßigen Abständen Arbeitsberichte heraus [34, 35].

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.

Tabelle 1. Veranstaltungen des JavaGrande-Forums

Workshop, Syracuse, Dezember 1996, [9]
PLDI Workshop, Las Vegas, Juni 1997, [10]
Java Grande Konferenz, Palo Alto, Februar 1998, [11]
EuroPar Workshop, Southampton, September 1998, [8]
Supercomputing, Ausstellung und Panel, November 1998, [34]
IEEE Frontiers 1999 Konferenz, Annapolis, Februar 1999
HPCN, Amsterdam, April 1999, [33]
IPPS/SPDP, San Juan, April 1999, [7]
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, [12]
JavaOne, Ausstellung und Panel, San Francisco, Juni 1999, [35]
ICS'99 Workshop, Rhodos, Juni 1999
Supercomputing, Ausstellung und Panel, September 1999
ISCOPE, JavaGrande Tag, Dezember 1999, [32]
IPPS, Cancun, Mai 2000
ACM Java Grande Konferenz, San Francisco, Juni 2000
Dagstuhl Seminar, August 2000

2. Warum Java für Hochleistungsrechnen?

Neben den üblichen Gründen, die auch im Fall „gewöhnlicher“ Anwendungen für die Verwendung von 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. Java bietet eine standardisierte Infrastruktur zur Realisierung verteilter Anwendungen auch im heterogenen Umfeld an. 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 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. Ein weiterer nicht unbedeutender Aspekt für die Anwender aus Wissenschaft und Ingenieurwesen ist, daß Java gelehrt und (begeistert) gelernt wird und eine breite Akzptanz hat, wohingegen bei Fortran ernster Nachwuchsmangel abzusehen ist.

2.1. Ablaufgeschwindigkeit und Speicherbedarf

Ein verbreitetes und zugleich hartnäckiges Gerücht ist, daß Java-Programme extrem langsam laufen. Die erste verfügbare Version von Java (1.0 beta) hat ByteCode interpretiert und war in der Tat extrem langsam. Seitdem hat sich 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 zwei Vergleichsergebnisse beurteilen den aktuellen Stand.

1. Vergleichendes Experiment. Prechelt [30] führte ein Experiment an der Universität Karlsruhe durch, bei dem das gleiche Problem mehrfach, von insgesamt 38 verschiedenen Personen gelöst wurde. Ziel war ein möglichst zuverlässiges Programm; das Experiment war nicht als Geschwindigkeitswettbewerb ausgelegt. Dabei entstanden 24 Java-, 11 C++- und 5 C-Implementierungen. Unter den 38 Personen, durchweg Informatik-Studenten im Hauptdiplom mit durchschnittlich 8 Jahren und 100 KLOC Programmiererfahrung, waren sehr gute bis relativ schlechte Programmierer.

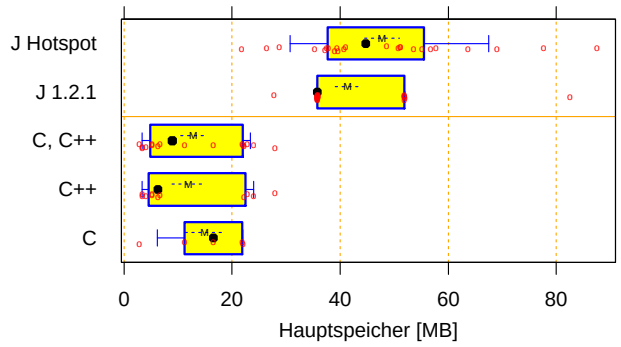

Abbildung 1. Auf Solaris 7 gemessener Hauptspeicherbedarf der untersuchten Programme (einschließlich Datenstrukturen, Programmcode, Bibliotheken, Prozeßverwaltung und JVM). Die einzelnen Messungen sind durch kleine Kreise symbolisiert. Das M zeigt jeweils den Mittelwert, der schwarze Punkt gibt den Median an. Innerhalb einer Box liegen die inneren 50% der Messungen, die H-Linie enthält die inneren 80% der Messungen.

Java (Version 1.2.1) braucht im Schnitt vier bis fünf mal soviel Hauptspeicher wie C/C++, siehe Abbildung 1. Die individuellen Unterschiede zwischen den Experimentteilnehmern waren gewaltig, und zwar bei allen Sprachen. Die beobachteten Laufzeiten variierten von Sekunden bis zu 30 oder gar 40 Minuten, siehe Abbildung 2. Es gibt aber 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, daß nur sehr wenige Datenpunkte vorliegen und daß vermutlich die besten Experimentteilnehmer C zur Realisierung ausgewählt haben.

Zusammengefaßt läßt sich sagen, daß C++ kaum noch einen Laufzeitvorteil gegenüber Java hat, daß die Programmiererunterschiede größer als die Sprachunterschiede sind, aber daß C++ noch immer weniger Hauptspeicher erfordert als Java.

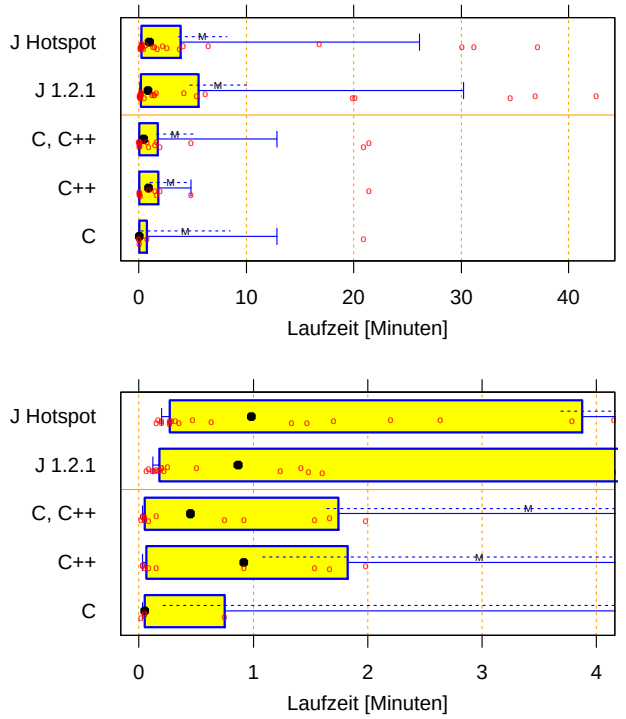

Abbildung 2. Auf Solaris 7 gemessene Laufzeiten der unterschiedlichen Programme; unten vergrößert dargestellt. Legende siehe vorherige Abbildung.

2. Benchmark für Wissenschaftsanwendungen. Pozo und Miller haben in SciMark 2.0 [29] fünf mittelgroße numerischen Kernroutinen kombiniert (schnelle komplexwertige Fouriertransformation, Gauss-Seidel-Relaxation, Monte-Carlo-Integration von e^{-x^2}, Multiplikation dünnbesetzter Matrizen und LU-Faktorisierung dichter Matrizen mit 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 auskommt.

Die veraltete JVM 1.1.5 des Netscape-Browsers erreicht auf einem Intel Celeron 366 unter Linux etwa 0,7 MFlops und ist damit 135 mal langsamer als die C-Implementierung. Java 1.1.8 hat stark aufgeholt: auf gleichem Prozessor werden (unter OS/2) derzeit etwa 76 MFlops erreicht – nur noch 35% weniger als mit C.

Insgesamt ist die Geschwindigkeit von Java besser als ihr Ruf; weitere Verbesserungen sind abzusehen. Zusätzlich finden sich immer häufiger 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 [24, 31]. 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, die durch die IFIP Working Group 2.5 (International Federation for Information Processing) unterstützt wird, 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 Abschnitten präsentieren konkrete Teilziele und die erreichten Resultate.

3.2. Verbesserung der Fließkomma-Arithmetik

Für wissenschaftliches Rechnen ist es wichtig, auf den meisten Prozessoren akzeptable Geschwindigkeiten und Genauigkeiten zu erreichen. Für Grande-Applikationen ist es darüberhinaus von Bedeutung, auf *manchen* Prozessoren sehr hohe Fließkomma-Leistung zu erreichen. Numeriker haben seit Beginn der Fließkomma-Arithmetik gelernt, mit architekturspezifischen Rundungsfehlern umzugehen, so daß eine exakte bitweise Reproduzierbarkeit *nur selten* von eminenter Bedeutung ist – ganz im Gegensatz zu Javas zentralem Entwurfsziel der exakten Reproduzierbarkeit der Ergebnisse. Nicht akzeptabel ist aber eine zu ungenaue Berechnung in der Mathe-Bibliothek (beispielsweise der trigonometrischen Funktionen), wie sie in vielen Java-Implementierungen bislang zu beobachten ist.

3.2.1. Fließkomma-Leistung

Um exakte Reproduzierbarkeit zu erreichen, verbietet Java gängige Optimierungen, wie z.B. die Assoziativität von Operatoren auszunutzen, weil dadurch evtl. eine Änderung des Rundungsverhaltens ausgelöst wird. Weitere Verbote beziehen sich auf Prozessoreigenschaften.

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 langsam gerundet werden, indem er aus dem Register in den Hauptspeicher geschrieben und wiedergeladen wird. Diese Rundung unterbleibt daher in vielen Implementierungen. Das zweimalige Runden (unmittelbar nach der Operation und bei der Interaktion mit dem Hauptspeicher) führt darüberhinaus in manchen Fällen zu Abweichungen von der Spezifikation in der Größenordnung von 10^{-324}. Eine Korrektur dieses Fehlers ist zeitaufwendig und unterbleibt ebenfalls in den meisten Implementierungen. Aufgrund beider Probleme ist Javas Fließkomma-Arithmetik auf Intel-Prozessoren entweder falsch oder 2 bis 10 mal langsamer als möglich.

2. Verwendungsverbot für „FMA – Fused Multiply Add“-Maschinenbefehl. Prozessoren der PowerPC-Familie bieten einen Maschinenbefehl an, der bei nur einer Rundung zwei Fließkommazahlen multipliziert und eine weitere Zahl dazuaddiert. Javas Sprachdefinition verbietet die Verwendung dieses Maschinenbefehls und verschenkt dadurch in manchen Experimenten über 55% der Leistung. Abbildung 3 stammt von José Moreira, IBM. Ohne alle Optimierung werden nur 3,8 MFlop erreicht. Wenn alle üblichen Optimierungen durchgeführt werden (einschließlich der Elimination redundanter Feldgrenzentests), dann erreicht Java mit dem nativen Java-Übersetzer der IBM 62% der Leistung eines äquivalenten Fortran-Programms (83,4 MFlop). Wenn zusätzlich der FMA-Maschinenbefehl ausgenutzt werden könnte (was der Fortran-Übersetzer routinemäßig tut), könnten fast 97% der Fortran-Leistung erreicht werden.

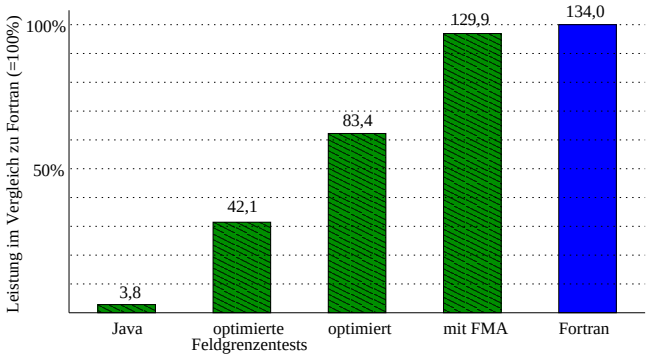

Abbildung 3. Cholesky-Faktorisierung auf 67 MHz Power-2-Prozessor. Angaben in MFlops und relativ zu optimiertem Fortran

3.2.2. 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. Im unattributierten Standardfall darf Java nun anonyme `double`-Variablen mit 15 Bit langen Exponenten darstellen und dadurch auf der x86-Familie auf die kostspieligen Speicher- und Ladeoperationen verzichten.

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.

3.2.3. Reproduzierbarkeit der `Math`-Funktionen

Neben den strikten Verboten, die die Fließkomma-Leistung beschränken, leidet Javas Fließkomma-Arithmetik auch noch unter einem Genauigkeitsproblem. Spezifikationsgemäß (Java 1.1.x) müssen für die Realisierung der Bibliothek `java.lang.Math` nach Java übertragene Implementierungen aus `fdlibm` verwendet werden.[2] Stattdessen verwenden viele Hersteller die von der Hardware angebotenen Maschinenbefehle, wodurch es zu schnelleren, aber inkorrekten Ergebnissen kommt. Um dieser Entwicklung entgegenzutreten, bietet Sun seit Java 2 Kapselungsmethoden für die in C realisierten `fdlibm`-Funktionen an und macht es leichter, die `fdlibm`-Ergebnisse auf allen Plattformen zu erzielen. Erst John Brophy, Visual Numerics, hat eine `fdlibm` vollständig in Java implementiert, so daß in Zukunft auf die Kapselungsmethoden verzichtet werden kann [1].

Eine ideale Mathe-Bibliothek würde Fließkomma-Zahlen liefern, die höchstens 0,5 ulp (unit in the last place) vom tatsächlichen Ergebnis entfernt 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. Abraham Ziv, IBM Haifa, hat eine Mathe-Bibliothek (in ANSI C) erstellt, die korrekt rundet (Fehler höchstens 0,5 ulp) [37].

3.2.4. Vorschlag `fastfp`-Schlüsselwort

Das JavaGrande-Forum erarbeitet derzeit einen JSR (Java Specification Request [21]), 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 die Verwendung des FMA-Befehls erzwingt. Auch 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.

In Java können komplexe Zahlen sinnvoll nur in Form einer `Complex`-Klasse realisiert werden, 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 komplexwertige Arithmetik langsamer als Javas primitivwertige Arithmetik, weil das Anlegen eines Objekts langsamer ist und mehr Speicherplatz verbraucht als das Anlegen einer Variable eines primitiven Typs und weil durch die Formulierung mit Hilfe von Methodenaufrufen Hilfsobjekte angelegt werden, die bei Verwendung der Stapelmaschine zur Wertspeicherung gar nicht erforderlich wären. (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.) 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 keine automatische Typkonvertierung auslöst. 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 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 Wertklassen erweitert wird, und da auch nach einer solchen Erweiterung Wertklassen nicht nahtlos in das Erscheinungsbild existierender primitiver Typen eingebunden sind, hält das JavaGrande-Forum das folgende zweigleisige Vorgehen für sinnvoll, siehe Abbildung 4.

3.3.1. Klasse `java.lang.Complex`

Das JavaGrande-Forum definiert eine Bibliothek `java.lang.Complex`, die sich im Stil an Javas andere Zahlenklassen anlehnt, und realisiert sie prototypisch. Zusätzlich sind arithmetische Operationen vorhanden, um methodenbasierte Operationen auf komplexen Zahlen ausdrücken zu können. Die Klasse wird Sun in Form eines JSR unterbreitet.

IBM hat die Semantik der `Complex`-Klasse fest in ihrem nativen Java-Übersetzer eingebaut; intern wird

[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.

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

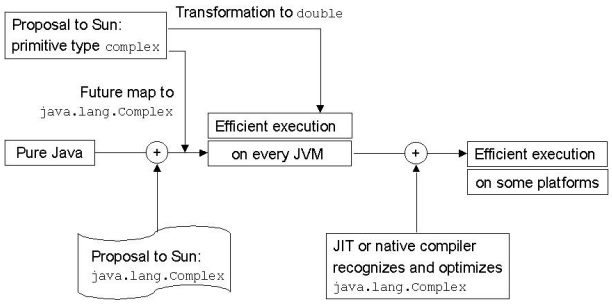

Abbildung 4. Einführung komplexwertiger Arithmetik

ein komplexer Datentyp verwendet und darauf werden die zugehörigen Optimierungen durchgeführt [36]. Die in Java-Notation vorhandenen Methodenaufrufe und Objekterzeugungen werden weitgehend entfernt bzw. über den Stapel abgewickelt, so daß man zumindest auf den unterstützten Plattformen von einer effizienten Unterstützung der Klasse `Complex` sprechen kann.

3.3.2. Primitiver Datentyp `complex`

Als weiteren JSR schlägt das JavaGrande-Forum einen primitiven Typ `complex` mit zugehörigen Infix-Operationen vor. Um weder das ByteCode-Format noch existierende JVM-Implementierung zu verändern, soll diese Spracherweiterung in einem Präprozessorschritt auf übliches Java zurückgeführt werden.

Abbildung 4 zeigt zwei Transformationen. 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 realisiert worden [13, 14]. 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 [23, 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 diverse „Zeilen“ durch ein gemeinsames eindimensionales Feld realisiert werden könnten (Alias), die „Zeilen“ unterschiedliche Länge haben könnten, jeder Zugriff auf mehrdimensionale Felder mindestens eine Indirektion zur Zeigerverfolgung erzwingt und schließlich bei jedem Feldzugriff zur Laufzeit überprüft wird, ob er innerhalb der Feldgrenzen liegt.

Obwohl oft durch Datenflußanalyse festgestellt werden kann, daß Feldzugriffe innerhalb der Feldgrenzen liegen, oder obwohl oft der Code so vervielfacht und mit Randtests 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 feste Ausrollreihenfolge können Algorithmen verwendet werden, 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 mehrdimensionale 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

Leichtgewichtige Klassen und Operator-Überladung sind die verallgemeinerte Lösung sowohl für komplexe Zahlen als auch für mehrdimensionale Felder. Leichtgewichtige Klassen haben Wertsemantik – ihre Instanzvariablen können also nach der Objekterzeugung nicht verändert werden. Daduch entfallen die Probleme der Gleichheitssemantik. Ferner können leichtgewichtige Objekte oft leistungssteigernd auf dem Stapel alloziert und über den Methodenaufrufstapel per Kopie übergeben werden. Wenn der Programmierer die Basisoperationen `+`, `-`, `*`, `/` und `[]` überschreiben und selbst implementieren kann, werden Präprozessor-Lösungen hinfällig.

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 „normalen“ 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. Deshalb und auch 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.

4. Arbeitsgruppe Parallelismus und Verteilung

4.1. Ziele der Arbeitsgruppe

Die Arbeitsgruppe Parallelismus und Verteilung des JavaGrande-Forums evaluiert die Verwendbarkeit von Java für paralleles und verteiltes Rechnen. Darauf aufbauend werden konsensfähige Aktionen konzipiert und ergriffen, um Unzulänglichkeiten der Programmiersprache bzw. des Laufzeitsystems zu beseitigen. Den konkreten Teilzielen und den jeweils erreichten Resultaten widmen sich anschließende Abschnitte. Weitere Arbeiten zu parallelen Programmierumgebungen und „Computing Portals“ sind noch nicht genügend konsolidiert und werden an dieser Stelle nicht vertieft.

4.2. Schneller entfernter Methodenaufruf

Für verteilte und parallele Programme sind gute Latenzzeiten und hohe Bandbreite essentiell.

Der entfernte Methodenaufruf gängiger Java-Implementierungen (RMI) ist 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 Argumente dauert ein entfernter Methodenaufruf mit Java Millisekunden, 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 die Übertragung (TCP/IP-Ethernet) entfällt.

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

An der Universität Karlsruhe wurden im Rahmen des JavaParty-Projekts [20] alle drei Aufgaben angegangen und die derzeit weltweit schnellste Java-Implementierung eines entfernten Methodenaufrufs gebaut. Auf einem Bündel von DEC-Alpha-Rechnern, die mit Myrinet-Karten verbunden sind, dauert ein entfernter Methodenaufruf derzeit vollständig[4] in Java etwa 80 μs. Abbildung 5 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 werden. Die zentralen Ideen der Optimierungen werden im folgenden beleuchtet.

[4] Die Anbindung des Kartentreibers ist nicht in Java realisiert.

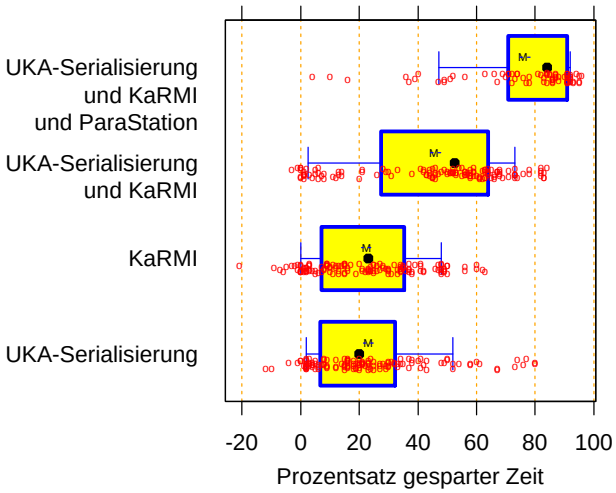

Abbildung 5. Die unteren drei „Zeilen“ zeigen je $2 \cdot 64$ Meßergebnisse für diverse Benchmarks (Ethernet auf PC und FastEthernet auf Alpha). Die unterste Zeile zeigt die erzielte Laufzeitverbesserung von RMI mit UKA-Serialisierung. Die zweite Zeile zeigt die Verbesserung, die KaRMI zusammen mit der JDK-Serialisierung bewirkt. Die darüberliegende Zeile zeigt die kombinierte Wirkung. Die oberste Zeile demonstriert das Verhalten, wenn zusätzlich zur UKA-Serialisierung und KaRMI noch Myrinet-Karten verwendet werden (64 Meßergebnisse).

4.2.1. UKA-Serialisierung

Die UKA-Serialisierung [15] kann statt der offiziellen Serialisierung (und als Ergänzung dieser) verwendet werden und spart 76%–96% der Serialisierungzeit. Sie basiert auf folgenden zentralen Ideen:

- Explizite Serialisierungsroutinen („Marshalling-Routinen“) sind schneller als das vom klassischen RMI verwendete automatische 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 für persistente Objektspeicherung nötig ist. Für Kommunikationszwecke, insbesondere in Rechnerbündeln mit gemeinsamem Dateisystem, ist eine schlankere Typverschlüsselung hinreichend und schneller. Das JavaGrande-Forum hat Sun Microsystems davon überzeugt, die Art der Typverschlüsselung in einer der nächsten Versionen wählbar zu machen.
- Kopierte Objekte sind in RMI bei jedem Aufruf erneut zu übertragen. RMI unterscheidet nicht zwischen Typverschlüsselung und Nutzdaten, so daß die Typinformation redundant übertragen wird.
  Sun hat angekündigt (bis jetzt ohne konkrete Versionsnennung), die Idee des getrennten Zurücksetzens von Typinformation und Nutzdaten aufzugreifen.
- Die offizielle Serialisierung nutzt mehrere Schichten von Strömen, die alle über eigene Pufferung verfügen. Dies führt zu häufigen Kopieroperationen und zu inakzeptabler Leistung. Die UKA-Serialisierung kommt mit einem Puffer aus, in den die Byte-Repräsentation unmittelbar geschrieben wird.

Sun hält an der Schichtung aus Gründen eines klareren objektorientierten Entwurfs fest, verbessert jedoch zumindest die Implementierung der Schichten.

4.2.2. KaRMI

In Karlsruhe entstand auch eine Ersatz-Implementierung des RMI. KaRMI [26, 28] kann statt des offiziellen RMI verwendet werden und beseitigt unter anderem die folgenden Unzulänglichkeiten im offiziellen RMI:

- KaRMI unterstützt Nicht-TCP/IP-Netzwerke. Aufgrund der Arbeiten des JavaGrande-Forums plant Sun dies (bislang nur schemenhaft) auch für die offizielle RMI-Version.
- KaRMI hat eine klare Schichtung, durch die es möglich wird, andere Protokollsemantiken (z.B. Multicast) und andere Netzhardware (z.B. Myrinet-Karten) zu verwenden.
- In RMI können Objekte an feste Portnummern gebunden werden, wodurch ein Detail der Netzwerkschicht an die Anwendung durchgereicht wird. Dies widerspricht dem Geheimnisprinzip. KaRMI unterstützt diese Funktionalität nur, wenn das unterliegende Netzwerk Portnummern anbietet.
- Im Gegensatz zu KaRMI ist das offizielle RMI nicht für Geschwindigkeit optimiert. Ausgelöst von den guten KaRMI-Ergebnissen hat Sun Personal zur Optimierung eingestellt.
- Der verteilte Speicherbereiniger des offiziellen RMI ist für Weitverkehrsnetze ausgelegt. Obwohl es für enggekoppelte Rechnerbündel (und andere Plattformen) optimierte Speicherbereiniger gibt [27], sieht das offizielle RMI im Gegensatz zu KaRMI keinen alternativen Speicherbereiniger vor.

4.3. Message Passing in Java

Während sich Javas Mechanismen für paralleles und verteiltes Programmieren am Client/Server-Modell orientieren, ist MPI ein symmetrisches, auf Botschaftenaustausch basierendes, paralleles Berechnungsmodell.

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:

- mpiJava: eine Sammlung von Kapselklassen, die über die native Schnittstelle von Java (JNI) auf die C++-Bindung von MPI zurückgreifen [4, 16].
- JavaMPI: automatisch für ein Programm erzeugte JNI-Kapselungen für die C-Bindung von MPI [25].
- MPIJ: eine Implementierung der MPI-Schnittstelle in Java, die sich an der C++-Bindung orientiert und recht gute Leistungsergebnisse vorzuweisen hat [22].

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, ob die in MPI verwendbaren Typen um Java-Objekte erweitert werden könnten, die dann in serialisierter Form verschickt würden [5]. Ferner wird geprüft, ob und wie Javas Thread-Modell verwendet werden kann, um die prozeßbasierten Ansätze von MPI 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) [6]. Derzeit existiert eine stabile Sammlung von nicht-parallelen Benchmarks in drei Kategorien:

- Basisoperationen (wie z.B. arithmetische Ausdrücke, Objekterzeugung, Methodenaufruf, Schleifenrümpfe etc.) werden vermessen.
- Berechnungskerne: Ähnlich wie bei SciMark werden numerische Kernroutinen betrachtet. In der Sammlung ist auch der IDEA-Verschlüsselungsalgorithmus.
- Applikationen: Zur Sammlung gehören eine Alpha-Beta-Suche mit Abschneiden, eine Computational-Fluid-Dynamics-Anwendung, die Simulation von Moleküldynamik, eine Monte-Carlo-Simulation und ein 3D-Strahlverfolgung.

Es wird an Thread-Benchmarks für alle drei Kategorien gearbeitet. Dabei werden 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 arbeitet man an einer möglichst vollständigen Menge von äquivalenten Implementierungen in C/C++, um einen Sprachvergleich zu ermöglichen.

5. Zusammenfassung

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 Präsentationsunterlagen von Geoffrey Fox, Dennis Gannon, Roldan Pozo und Bernhard Haumacher. 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. Implementation of `java.lang.complex`. http://www.vni.com/corner/garage/grande/.
2. C9x proposal. http://anubis.dkuug.dk/jtc1/sc22/wg14/ and ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/.
3. B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: 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, June 12–14, 1999.
6. Java Grande Benchmarks. http://www.epcc.ed.ac.uk/javagrande
7. J. Rolim et al., editor. *Parallel and Distributed Processing*. Number 1586 in Lecture Notes in Computer Science. Springer Verlag, 1999.
8. *Proc. Workshop on Java for High Performance Network Computing at EuroPar'98*. Southampton, September 2–3, 1998.
9. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(6). John Wiley & Sons, June 1997.
10. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(11). John Wiley & Sons, November 1997.
11. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 10(11–13). John Wiley & Sons, September–November 1998.
12. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, to appear. John Wiley & Sons, 2000.
13. Edwin Günthner and Michael Philippsen. Komplexe Zahlen für Java. In *JIT'99, Java-Informations-Tage*, pages 253–266, Düsseldorf, September 20–21, 1999. Springer Verlag.
14. Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, to appear, 2000.
15. 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.
16. http://www.npac.syr.edu/projects/pcrc/HPJava/.
17. Java Grande Forum. http://www.javagrande.org.
18. Java Grande Forum, mailinglist. All Members: javagrandeforum@npac.syr.edu, Subscribe: gcf@syracuse.edu.
19. Java Grande Forum, Europe. http://www.irisa.fr/EuroTools/Sigs/Java.html.
20. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
21. The Java community process manual. http://java.sun.com/aboutJava/communityprocess/java_community_process.html.
22. 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, June 12–14, 1999.
23. 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.
24. Reinhard Klemm. Practical guideline for boosting Java server performance. In *ACM 1999 Java Grande Conference*, pages 25–34, San Francisco, June 12–14, 1999.
25. S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Wańiewski, editors, *Recent Advances in PVM and MPI*, Lecture Notes in Computer Science, pages 135–142. Springer Verlag, 1997.
26. 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. Springer Verlag.
27. Michael Philippsen. Cooperating distributed garbage collectors for clusters and beyond. *Concurrency: Practice and Experience*, to appear, 2000.
28. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, to appear, 2000.
29. Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark/.
30. Lutz Prechelt. Comparing Java vs. C/C++ efficiency differences to inter-personal differences. *Communications of the ACM*, 42(10):109–112, October 1999.
31. Mark Roulo. Accelerate your Java apps! *Java World*, September 1998.
32. 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, 1999.
33. 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, 1999.
34. 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.
35. 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.
36. 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, June 12–14, 1999.
37. Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, (17):410–423, 1991.