Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java

Michael Philippson, Matthias Jacob und Martin Karrenbach

1 Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Am Fasanengarten 5, 76128 Karlsruhe
2 Universität Karlsruhe, Geophysikalisches Institut, Hertzstrasse 16, 76128 Karlsruhe


Schlüsselwörter: Wissenschaftliches Rechnen in Java, parallele seismische Methoden, Veltran, Vergleich von Java mit Fortran90/HPF

Summary. Java is often accused of being too slow for serious programming, especially for scientific problem solving. However, we found that for large-scale geophysical application, Java code compiled with current just-in-time compilers runs slower than Fortran by a factor of at most 4, on both a shared-memory parallel machine (SGI Origin2000) and a distributed-memory parallel machine (IBM SP/2). The slowdown is easily offset by the following advantages: (a) object-oriented Java code is easier to maintain and reuse than Fortran code, (b) Java code is fully portable, even among parallel computers with different memory models. Furthermore, better compiler technology is on the horizon, which will narrow the performance gap even more.

Key words: Computational Science in Java, parallel seismic methods, Veltran, comparison of Java versus Fortran90/HPF


1. Einleitung


In der Erdölindustrie ist es üblich, entweder in Fortran77 zu implementieren und Message Passing Bibliotheken wie PVM oder MPI für die Verteilung zu verwenden, oder bereits in Fortran90 bzw. HPF zu programmieren. Diese Ansätze haben mehrere Nachteile, die insbesondere dann ins Gewicht fallen, wenn neuartige Lösungsstrategien erprobt werden und die Arbeitszeit der Entwickler die kritische Ressource ist.

Bei der Verwendung von Message Passing Bibliotheken zur Implementierung seismische Methoden in parallelen Systemen mit verteiltem Speicher muß der Programmierer über die geophysikalischen Probleme hinaus die Parallelisierung selbst vornehmen, wobei Lastenausgleich und Lokalität von entscheidender Bedeutung sind. Der resultierende Programmcode ist nicht nur fehleranfälliger, sondern auch schwer zu portieren.

Insbesondere für neue Lösungsstrategien werden deshalb in der Erdölindustrie neuerdings Fortran90 oder HPF bevorzugt, die jedoch wie alle prozeduralen Programmiersprachen unter eingeschränkter Wiederverwendbarkeit von Software leiden. Ferner ist es schwierig, den
Quellcode zu warten und eine natürliche Hierarchie von Operatoren aufzubauen, wie sie für eine Architektur von
seismischen Methoden vorteilhaft wäre.

Die Programmiersprache Java [2, 3] hingegen ist objekt-orientiert und umfaßt Sprachelemente, mit deren
Hilfe parallele Programme auf Basis von Threads und Synchronisationspunkten ohne zusätzliche Systembibli-
otheken ausgedrückt werden können. Die in Java integrierte
RMI-Kommunikationsschnittstelle (Remote Method Invocation) vereinfacht den Datenaustausch zwischen
den einzelnen Prozessoren. Dies ermöglicht Parallelverar-
beitung auf Anwendungsgebrauch in der Java-Umgebung
selbst, wie es Hassanzadeh et al. für Geophysik [4] er-
örtern. Somit besteht kein weiterer Bedarf für Message
Passing oder virtuellen gemeinsamen Speicher in der
Anwendung. Darüberhinaus ist dieser parallele Ansatz
hardwareunabhängig; die Programmieran auf verschie-
denen Systemplattformen.

Aus der Sicht der Softwaretechnik scheint Java al-
so eine geeignete Programmiersprache für die parallele
Implementierung von seismischen Methoden zu sein. In
dieser Fallstudie wird gezeigt, daß Java zumindest für
eine Reihe von geophysikalischen Basalgorithmen eine
akzeptable Leistung beim wissenschaftlichen Rechnen er-
reichen kann.

Für diese Zwecke wurden einige geophysikalische Ba-
salgorithmen untersucht. Wegen der insgesamt gleich-
lautenden Ergebnisse konzentriert sich dieser Artikel auf
einen Operator für die Geschwindigkeitsanalyse, der Vel-
tran genannt wird und eine fundamentale Methode zur
Verarbeitung seismischer Daten ist. Die Implementierung
basiert auf einer vorläufigen Version von JAG, einer
Architektur seismischer Operatoren in Java [9, 11], an
der im Stanford Exploration Project momentan gear-
beitet wird. Nach einer kurzen Einführung in den geo-
physikalischen Hintergrund im zweiten Abschnitt, wird
in Abschnitt 3 JavaParty, ein System zur vereinfachte-
en verteilten Programmierung in Java, erläutert. Ab-
chnitt 4 diskutiert die parallele Implementierung des
Operators, bevor anschließend die Meßergebnisse disku-
tiert werden.

2. Der Veltran Operator

Veltran ist ein geophysikalischer Operator für die Ana-
lyse der spezifischen Geschwindigkeiten der inneren Erschich-
enden und wird gewöhnlich als erstes Glied einer Kette von Operatoren eingesetzt, an deren Ende ein
Profil der Erdschichten produziert wird. Veltran verar-
beitet die rohen Meßdaten, die von seismographischen
Empfängern aufgezeichnet wurden.

In Abb. 1 wird eine sich zirkular ausbreitende Welle
an Punkt s erzeugt, wobei ein Teil der eintreffenden
Welle reflektiert wird, wenn sich die Medien zweier hori-
Zontaler Schichten l1 und l2 in ihrer spezifischen Ge-
schwindigkeit unterscheiden. Der reflektierte Teil dieser
Welle wird von den seismographischen Empfängern x1-
x7 an der Erdoberfläche gemessen. Auf Basis dieser Meß-
daten berechnet der Veltran Operator die spezifischen
Geschwindigkeiten der Erdschichten, wobei ein verein-
fachtes Modell der Erde angenommen wird, in dem die
Schichten horizontal angeordnet sind.

Am Meßpunkt x4i, der die Entfernung x1-s vom Erzeu-
gungspunkt besitzt, wird über die Signalanlaufzeit t die
Signalkomponente aufgezeichnet. Jedes der Diagramme in Abb. 2,
 die Quellmatrizen genannt werden, zeigt diese gemesse-

Abbildung 1. Messungen seismischer Daten – die Wellen entste-
en in s; x0-x7 sind unabhängige seismographische Empfänger.
Die gestrichelte Linie bezeichnet die Grenze zwischen zwei hori-
zontalen Erdschichten l1 und l2 mit unterschiedlichen spezifischen
Geschwindigkeiten.

Abbildung 2. Quellmatrizen von seismischen Meßdaten: (a) syn-
thetische Daten, (b) Felddaten [Mobil Oil]. „Time“ bezeichnet die
Signalanlaufzeit t (siehe unten) und „Distance“ die Entfernung von
s zu x1.
nen Signalstärken als Grauwerte in Abhängigkeit von der Signallaufzeit und der Entfernung des Empfängers vom Erzeugungspunkt. Je heller ein Punkt erscheint, desto größer ist die aufgezeichnete Signalstärke. Eine Quellmatrix hat für jeden der \( n_x \) Empfänger eine Spalte; die Anzahl der Zeilen (\( n_y \)) ergibt sich durch die Abtastrate.

Mathematisch betrachtet können die Linien beliebig aussehen, bei einem horizontalen Schichtenmodell entsprechen sie jedoch Hyperbeln. Das obere Diagramm in Abb. 2 verwendet synthetische Daten, um die Hyperbeln zu veranschaulichen, wohingegen das untere aus realen Daten hervorgegangen ist.

Wenn nun der Zeit-Offset \( \tau \) bekannt wäre, der der Zeit für die Signalsausbreitung von \( s \) senkrecht nach unten zur Schicht und wieder zurück entspricht, könnte die „normal moveout“-Formel \( t = \sqrt{\tau^2 + \frac{s^2}{v^2}} \) zur Bestimmung der Geschwindigkeiten benutzt werden.

Leider kann \( \tau \) nicht gemessen werden, da es nicht möglich ist, einen Empfänger direkt an \( s \) zu plazieren, und somit muß Veltran die spezifischen Geschwindigkeiten und die Tiefe der Schichten aus den Mßdaten ableiten.

Die Berechnung wird nun folgendermaßen durchgeführt: Veltran iteriert potentielle Geschwindigkeiten aus einem Bereich von \( n_y \) Werten. Für jede dieser Geschwindigkeiten werden hypothetische Hyperbeln berechnet, danach werden die Signalmessungen aus der Quellmatrix entlang dieser Hyperbeln aufsummiert. Für diejenige hypothetische Hyperbel, die mit der Hyperbel innerhalb der Quellmatrix übereinstimmt, erreicht die Summe ein Maximum und nähert daher die gesuchte Geschwindigkeit am besten an. Die Berechnung der Summe benötigt \( n_y \) Schritte, da jeder Zeitpunkt während des Experiments betrachtet werden muß. Daher benötigt Veltran für eine einzelne Quellmatrix \( n_x \cdot n_y \cdot n_y \) Schritte.

Da die Lage der Hyperbeln unbekannt ist, werden alle Punkte der Quellmatrix unterschiedlich häufig bei der Iteration erfaßt. Der iterative Ansatz wird durch die irregulären Datenabhängigkeiten erforderlich.

Abb. 3 zeigt das Ergebnis von Veltran: Je heller ein Punkt für eine angenommene Geschwindigkeit ist, desto größer ist die Summe der gemessenen Signalstärken entlang der Hyperbel, wobei sich der hellste Punkt in jeder Zelle auf die spezifische Geschwindigkeit der Schicht mit dem Zeit-Offset \( \tau \) bezieht. Die Ergebnismatrix beschreibt also die Situation auf einer Linie, die vom Erzeugungspunkt \( s \) aus ins Erdinnere verläuft (eindimensionales Bild des Erdinneren). Einige unvermeidbare Seiteneffekte, die durch mehrfache Reflektionen verursacht werden, wie sie z.B. auch in Abb. 1 gezeigt sind, lassen oft mehr als einen Punkt hell erscheinen. Um die Stärke solcher Effekte abzuschwächen, werden zusätzliche Techniken verwendet, die über den Rahmen dieses Artikels hinausgehen.

Zur Konstruktion eines zweidimensionalen Bildes des Erdinneren werden Messungen für mehrere Erzeugungspunkte durchgeführt, die sich auf eine Linie an der Erdoberfläche befinden. Das resultiert in einer Menge von Quellmatrizen, die zu einem dreidimensionalen Würfel angeordnet werden. Zusätzlich zu den \( n_x \cdot n_x \cdot n_y \) Schritten, die Veltran auf einer einzelnen Quellmatrix benötigt, muß nun die dritte Dimension, bestehend aus \( n_x \) Elementen, berücksichtigt werden.


3. JavaParty – Parallele und Verteilte Programmieren in Java


Auf diesen Systemplattformen besteht ausschließlich die Möglichkeit, entweder explizierte Socket-Instruktionen oder RMI zur Kommunikation zwischen den Knoten zu verwenden. Beide Ansätze resultieren jedoch oft in komplizierten und fehleranfälligen Programmen, die allgemein ähnliche Nachteile wie die Programmierung mit MPI aufweisen:


Es bleibt den Entwicklern überlassen, die Entscheidungen anstelle des ObjectDistributors selbst zu treffen und manuelle Verteilungsstrategien zu implementieren. Analog kann die automatische Objektmigration abgeschaltet und durch explizite Migrationsinstruktionen ersetzt oder ein anwendungspezifischer ObjectDistributor und MigrationManager eingesetzt werden. Die Implementierung des Veltran Operators, die im folgenden diskutiert wird, benutzt den normalen ObjectDistributor von JavaParty.

Der JavaParty Präprozessor und die übrige JavaParty Umgebung bestehen zu 100% aus Java und sind frei verfügbar [7].

4. Parallelisierung von Veltran

Veltran erfordert wegen seiner langen Laufzeit und seines Speicherbedarfs eine effiziente Parallelisierung.

Wie oben bereits angedeutet wurde, kann die Geschwindigkeitsanalyse mehrerer Matrizen unabhängig von einander und trivial parallel durchgeführt werden. Um jedoch die Anzahl der verfügbaren Prozessoren auszunutzen, werden im folgenden die Parallelisierungsmöglichkeiten einer einzelnen Quellmatrix für einen einzelnen Erzeugungspunkt betrachtet.

Das Hauptproblem bei der Parallelisierung von Veltran ist die nicht-lineare Struktur der Hyperbolen, die darin besteht, daß für eine hypothetische Geschwindigkeit $v_1$ die während der Iterationen von Veltran untersucht wird, potentiell alle Werte aus dem $t \times x$ Feld in die Berechnung des Maximums, d.h. der Geschwindigkeit, einbezogen werden könnten.

Bei einer Parallelisierung durch Aufteilung der zu verrichtenden Arbeit entlang der $t$- oder $x$-Achse (siehe Abb. 4) muß die Matrix auf Maschinen mit erteiltem Speicher repliziert werden und jeder Prozessor muß eine Matrix von Teilmatten berechnen, die im Anschluß aufsummiert werden, um das Endergebnis der Geschwindigkeitsanalyse zu erhalten. Diese Summation erfordert einen zusätzlichen Aufwand von $n_t \times n_x$ Operationen.

Parallelisierungen entlang der $r$- oder $v$-Achsen (siehe Abb. 5) kommen mit reduzierter Datenreplikation aus
und benötigen keine abschließende Summation. Sie sind daher effizienter. In beiden Fällen berechnet jeder Prozessor einen Streifen der Zielmatrix, die zum Abschluß lediglich konkateniert werden müssen.


Nachdem die Quellmatrix unter den Prozessoren aufgeteilt wurde, startet der DataDistributor einen Thread pro entferntem Prozessor, um Lastenausgleich unter den Prozessoren zu erzielen.

Im Fall der Berechnung mehrerer Quellmatrizen entlang der $x$-Achse, wird die optimale Leistung erreicht, falls die Matrizen auf Gruppen von Prozessoren verteilt werden, wobei jede einzelne Gruppe eine Matrix parallel bearbeitet. Unsere Experimente haben gezeigt, daß die Parallelisierung entlang der $v$-Achse die besten Laufzeiten bringt.

Abb. 4. Auflösung der Arbeit entlang der $x$- oder $t$-Achse. Zu erkennen ist die Replikation der Daten der Quellmatrix und die zusätzliche Summation.

Abb. 5. Auflösung der Arbeit entlang der $t$- oder $v$-Achse. Nur die schattierten Bereiche der Quellmatrix müssen auf Parallelrechnern mit verteiltem Speicher repliziert werden. Die Zielmatrix ergibt sich durch Konkatenation der Streifen, die die Prozessoren eigenständig berechnen.

5. Versuchsauflauf


Abb. 7. Technische Daten der Maschinen.

<table>
<thead>
<tr>
<th>SGI Origin2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: MIPS R10000 Prozessor Revision: 2.6</td>
</tr>
<tr>
<td>FPU: MIPS R10010 Gleitkomma-Prozessor Rev. 0.0</td>
</tr>
<tr>
<td>Größe des 2. Instruktionen-/Datenspeicher: 4 MBytes</td>
</tr>
<tr>
<td>Datenspeichergöße: 32 KBytes</td>
</tr>
<tr>
<td>Instruktionspeichergöße: 32 KBytes</td>
</tr>
<tr>
<td>Hauptspeichergöße: 3072 MBytes</td>
</tr>
<tr>
<td>Betriebssystem: Irix 6.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IBM SP/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 66MHz R8/6000 Prozessoren</td>
</tr>
<tr>
<td>Datenspeichergöße: 128 KBytes</td>
</tr>
<tr>
<td>Instruktionspeichergöße: 32 KBytes</td>
</tr>
<tr>
<td>Hauptspeichergöße: 64 MBytes pro Knoten</td>
</tr>
<tr>
<td>Betriebssystem: AIX 4.1</td>
</tr>
</tbody>
</table>


Auf der SGI wurde JDK 1.1.5 mit „just-in-time“-Übersetzer verwendet, da erst diese Version in der Lage ist, Threads tatsächlich parallel auszuführen. Auf der SGI wurde der standardmäßige Fortran90 Übersetzer, ebenfalls mit eingeschalteter Optimierung genutzt.

6. Ergebnisse


Abb. 8 zeigt die Laufzeit von JavaParty-Code gegenüber High-Performance Fortran-Code (HPF) auf der IBM SP/2. Die Java-Party-Version des parallelen Veltran Operators ist etwa um Faktor 3 langsamer als optimiertes HPF.


Die Verlängerung ist in beiden Fällen zum Teil darauf zurückzuführen, daß Java zur Laufzeit jeden Arrayzugriff auf zulässige Indexgrenzen überprüft.

Die Java-Programme passen sich automatisch der zur Verfügung stehenden Prozessorzahl und der Zahl der zu bearbeitenden Matrizen an. Den Fortran-Programmen fehlt diese Anpassungsfähigkeit, die aufgrund eingeschränkter Ausdrucksfähigkeit der Übersetzerdirektiven nicht angedacht werden kann, d.h. für jede Messung wurden manuell Konstanten im Programm geändert und neu übersetzt. Häufig das allgemeinsten Programm für alle Messungen verwendet werden müssen, dann hätten alle Fortran-Messungen dieselbe Laufzeit gezeigt. Beispielsweise hätte die SP/2 zur Berechnung von einer Matrix auf 4 Knoten ebenfalls 8 Sekunden benötigt (3 Sekunden im Fall der SGI).

Maschinencode erzeugen. Diese Übersetzer werden noch weiter an die gemessene Fortran-Leistung herankom-
men, da bessere Optimierungstechniken verwendet wer-
den können als bei „just-in-time“-Übersetzung.

7. Zusammenfassung

In dieser Studie wurde ein Verlangsamsungs faktor von unter 4 im Vergleich zu Fortran90 bzw. HPF bei der
parallelen Implementierung eines fundamentalen geo-
physikalischen Operators in Java erreicht. Als Testplatt-
dfment dienten zwei verbreitete Systeme für wissenschaft-
lches Rechnen – je ein paralleler Rechner mit gemeinsa-
men und mit verteilter Speicher. Ähnliche Ergebnisse
konnten für weitere seismische Operatoren und auf an-
deren Plattformen erwartet werden. Der Faktor 4 ist
akzeptabel, weil objekt-orientierter Java-Code leichter
t zu warten und wiederzuverwenden ist als Fortran-Code,
Java-Code komplett portabel ist, sogar zwischen verschiede-
denen Plattformen und die Leistung von Java sich weiter
verbessern wird, da Forschung im Bereich der Basis-
technologie von Java gegenwärtig verstärkt und mit erheblich
finanziellem Aufwand betrieben wird.

Danksagungen

Wir danken der JavaParty-Gruppe, besonders Matthias
Zenger, für ihre Unterstützung bei der Entwicklung und
dem Einsatz des JavaParty-Systems. Matthias Schab
vom Stanford Exploration Project stellte uns seine Ar-
chitektur für geophysikalische Operatoren zur Verfügung.
Weiterhin danken wir dem Maui High Performance Com-
puting Center und dem Rechenzentrum Karlsruhe für
den Zugang zur IBM SP/2 bzw. der SGI Origin2000.

Literatur

1. Eigenmann, R., Hassanzadeh, S.: Benchmarking with Real
Industrial Applications: The SPEC High-Performance Group,
IEEE Computational Science and Engineering, Spring issue,
3(1):18–23 [1996]

2. Gosling, J., Joy, B., Steele, G.: The Java Language Specifi-
cation. Reading, MA: Addison-Wesley 1996

3. Hassanzadeh, S., Mosher, C.C.: Java: Object-Oriented pro-
grressing for the cyber age. The Leading Edge, 15(12):1379–
1381, Dezember 1996

4. Hassanzadeh, S., Mosher, C.C., Joyner, C.L.: Scalable paral-
ellel seismic processing. The Leading Edge, 15(12):1363–1366,
Decembar 1996

5. Holt, C., Singh, J.P., Hennessy, J.: Application and Architec-
tural Bottlenecks in Large-Scale Distributed Shared Memory
Machines. In: 23th Int. Symp. on Computer Architecture, 134–
145, 1996

6. IBM: High Performance Compiler for Java.
http://www.alphaWorks.ibm.com
and Performance Portability on Shared Virtual Memory and
Hardware-Coherent Multiprocessors. In: 6th ACM SIGPLAN
Symp. on Principles and Practice of Parallel Programming
9. Nichols, D., Urdaneta, H., Oh, H.I., Clearbout, J., Laane, L.,
Karrenbach, M.: Programming geophysics in C++. In: Stan-
ford Exploration Project Report No. 79 [1993]
10. Philippens, M., Zenger, M.: JavaParty – Transparent Re-
 mote Objects in Java. Concurrency: Practice and Experience,
11. Schwab, M., Schroeder, J.: A seismic inversion library in Ja-
http://sepwww.stanford.edu/sep/jag
