UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Diplomarbeit

Parallele Schleifenprogramme in Java

Aufgabensteller:
Prof. Christian Lengauer Ph.D.
Lehrstuhl für Programmierung

Betreuer:
Dr. Martin Griebl

Autor:
Andreas Wolf

4. April 2000

Vorbemerkungen

Danksagungen Zunächst möchte ich Herrn Prof. Christian Lengauer für die interessante Themenstellung danken, und dafür, daß er mich ermutigt hat, diese Arbeit trotz meines anfänglichen Zögerns zu beginnen. Mein besonderer Dank gilt Martin Griebl für die viele Zeit und sicherlich auch Nerven, die er in die Diskussionen mit mir investiert hat. Sein Rat und seine Unterstützung haben wesentlich zum Gelingen dieser Arbeit beigetragen. Für die freundliche Unterstützung per Mail und Post danke ich außerdem Aart Bik und Bryan Carpenter, die für zwei der untersuchten Projekte verantwortlich sind. Vielmals danke ich auch den Mitarbeitern des Lehrstuhls: Peter Faber und Christoph Herrman für Ihre Anregungen und das "Querlesen" von Forschungspapieren. Allen Beteiligten danke ich für das gute und freundschaftliche Klima. Es war immer ein großes Vergnügen mit ihnen zusammenzuarbeiten.

Bemerkungen zur Typographie Diese Arbeit wurde mit LaTeX 2$_\varepsilon$unter Verwendung des $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX-Pakets erstellt. Die Abbildungen wurden mit `xfig`, die Diagramme mit StarOffice für Linux angefertigt.

Zum besseren Verständnis wurden einige Begriffe durch besonderen Zeichensatz hervorgehoben:

- Soweit nicht anders gekennzeichnet, beschreiben Ausdrücke in `Schreibmaschinen-Schrift` Schlüsselwörter in der betrachteten Programmiersprache, allgemein bekannte Programm-Konstrukte wie `for` oder `forall`, sowie die Namen ausführbarer Programme.

- Zentrale Begriffe, ebenso wie wichtige Eigennamen werden an der Stelle ihrer ersten Erwähnung *kursiv* gedruckt. Im weiteren Verlauf werden diese nicht mehr gesondert hervorgehoben.

- Im Index kennzeichnen **fett** gedruckte Seitenzahlen wie üblich die Seite der Definition bzw. das zentrale Auftreten des Begriffes (des Namens, des Schlüsselwortes etc.).

Inhaltsverzeichnis

1. Einleitung

Die Diplomarbeit soll die Eignung von neu entwickelten Java-Dialekten für die effiziente Implementierung von parallelen Schleifenprogrammen untersuchen. Dabei geht es darum, einige charakteristische Fallstudien zur Erprobung und zum Vergleich verschiedener Java-Dialekte vorzunehmen.

Seit SUN 1995 erstmals Java der Öffentlichkeit präsentierte, ist die Begeisterung für diese Sprache ungebrochen. Waren es anfangs vor allem Webdesigner, die die Plattformunabhängigkeit zu schätzen wußten, so gibt es mittlerweile kaum einen Bereich, den Java noch nicht für sich erschlossen hat. Neben javafähigen Autos[1] und (demnächst) Haushaltsgeräten sind Java-Programme auf nahezu jeder Plattform ausführbar. Nicht in erster Linie die Effizienz, sondern die massive Verfügbarkeit machen Java für das Parallelrechnen attraktiv. In Veröffentlichungen ([24] und enthaltene Literaturverweise) findet sich bereits die Idee des Internet-Computers mit Millionen einzelner Knoten.

Doch nicht nur die Verfügbarkeit, auch das Design der Sprache macht Java für viele Anwendungen attraktiv (genaueres findet sich in Kapitel 2). Die integrierten Sicherheitskonzepte erlauben die kontrollierte Ausführung von fremden Programmen auf dem eigenen Rechner. Die strenge Typisierung, die dynamische Bindung und ein detailliertes Laufzeit-Typsystem sowie die implizite Konsistenzprüfung bei allen Zeiger-Operationen senken die Entwicklungszeiten und erleichtern den Umgang mit komplexen Projekten.

Schließlich hilft der (im Vergleich etwa zu C++) geringe Umfang der Basissprache beim Erlernen der Sprache und vereinfacht die Entwicklung neuer Compiler. Eine große Zahl an Bibliotheken, die mit zum Standardpaket gehören, erleichtern nicht zuletzt die Netzwerkprogrammierung und die Entwicklung verteilter Anwendungen.

Vor diesem Hintergrund ist es beinahe natürlich, daß sich viele Forschungsgruppen der Erweiterung von Java für den wissenschaftlichen Bereich verschrieben haben. Neben verschiedenen Dialekten gibt es optimierende (und parallelisierende) Compiler sowie Kommunikationsbibliotheken und Schnittstellen zu Standardbibliotheken wie MPI. Grundsätzlich lassen sich bei allen Erweiterungen zwei verschiedene Richtungen erkennen, die mit *Dialekt oder Bibliothek* kurz und treffend charakterisiert sind.[2]

- Der erste Ansatz benutzt das sichere Design und den geringen Umfang der Sprache Java als Basis für die Definition eigener Dialekte und stellt zudem die nötigen Compiler bereit, um die neue Sprache direkt in Maschinencode zu übersetzen.[3] Dadurch

[1] Näheres dazu findet sich unter `http://java.sun.com/features/1999/06/concept_car.html`

[2] Selbstverständlich gibt es auch zahlreiche Mischformen, diese lassen sich jedoch meist nach ihrem Kerngedanken einer der beiden Richtungen zuordnen.

[3] "Klassische" Java-Compiler übersetzen das Quellprogramm in einen maschinenunabhängigen Bytecode.

ergibt sich zunächst ein potentieller Geschwindigkeitsvorteil für die Programmausführung. Zudem besteht die Möglichkeit, direkt (d.h. ohne Umwege über eine zusätzliche Java-Schnittstelle) auf bestehende Kommunikationsbibliotheken (z.B. MPI...) aufzusetzen.

- Der zweite Ansatz erweitert die Funktionalität mittels Bibliotheken bzw. realisiert geeignete Java-nach-Java Precompiler. Dadurch bleibt die Plattformunabhängigkeit der Zielsprache weitgehend[4] erhalten und die Zielprogramme lassen sich auf jeder javafähigen Plattform ausführen. Zudem steht diesen Programmen die gesamte Funktionalität der (mittlerweile beachtlichen) Java Standardbibliotheken zur Verfügung.

Gliederung dieser Arbeit

Kapitel 2 gibt einen groben Überblick über den Java-Standard. Da eine umfassende Betrachtung den Rahmen sprengen würde[5], werden lediglich einige Aspekte hervorgehoben, die für die weiteren Untersuchungen von Bedeutung sind. Besondere Aufmerksamkeit wird einigen Eigenschaften von Java gewidmet, die sich im Hinblick auf das Hochleistungsrechnen möglicherweise vor- bzw. nachteilig auswirken und somit für Projekte in diesem Bereich von Interesse sein können.

In den Kapiteln 3, 4, 5 und 6 werden einige konkrete Projekte detailliert untersucht, die verschiedene Herangehensweisen repräsentieren.

- *Titanium* (Kapitel 3) benutzt Java wegen seines Designs als Ausgangssprache und setzt im übrigen auf Maschinencode und hardwarenahe Kommunikationsbibliotheken.

- *HighPerformanceJava* (Kapitel 4) bevorzugt Java wegen seiner Plattformunabhängigkeit als Zielsprache und bietet Ansätze zur automatischen Parallelisierung von Bytecode.

- *HPJava* (Kapitel 5) verbindet beide Ansätze, setzt auf datenparallele Spracherweiterungen zur einfacheren Steuerung von Parallelität und Java als plattformunabhängige Zielsprache.

- *JavaParty* (Kapitel 6) ist vollständig in Java realisiert und ermöglicht den transparenten Zugriff auf entfernte Objekte. Dazu wird der Sprachumfang von Java um das Schlüsselwort `remote` erweitert, welches Klassen kennzeichnet, die von JavaParty netzwerkfähig gemacht werden sollen. Die Laufzeitumgebung bietet von allen untersuchten Projekten die einzige lauffähige Version einer verteilten virtuellen Maschine.

[4] Einige Erweiterungen, die das JNI (Java-Native-Interface) verwenden, benötigen zusätzlich Bibliotheken in Maschinencode.

[5] Der interessierte Leser sei an dieser Stelle auf die umfangreiche Literatur zu diesem Thema verwiesen (u.a. [5, 30]).

Als Grundlage für eine praktische Bewertung wurden drei Algorithmen mit den zu untersuchenden Sprachen implementiert. Die dabei gemachten Erfahrungen finden sich zusammen mit einer Beschreibung der Algorithmen in Kapitel 7. Wichtige Punkte bei der Untersuchung waren Ausdrucksstärke und Anwendbarkeit, insbesondere in Bezug auf die Parallelisierung von Schleifenprogrammen. Die ebenfalls in Kapitel 7 enthaltenen Zeitmessungen sind nur bedingt zum Vergleich der Sprachen untereinander geeignet[6].

Kapitel 8 enthält schließlich eine Auswahl von Forschungsarbeiten zu verwandten Themen. Wegen der Vielzahl von Projekten, die sich mit Erweiterungen zu Java (insbesondere für das Hochleistungsrechnen) beschäftigen, kann diese Liste natürlich nur unvollständig sein.

Anmerkungen zu einigen Begriffen

Wenn in dieser Arbeit vom Java-Standard gesprochen wird, ist damit das von SUN Microsystems veröffentlichte Java™2 Software Development Kit, Standard Edition Version 1.2 bzw. die Vorgängerversion 1.1 gemeint[7]. Dieses wurde im Mai 1999 der ECMA zur Standardisierung vorgelegt. Sun, Sun Microsystems, Solaris, Java, Java Virtual Machine sowie StarOffice sind eingetragene Warenzeichen von Sun Microsystems. Alle anderen in dieser Arbeit benutzten Namen und Bezeichnungen sind möglicherweise ebenfalls Warenzeichen der entsprechenden Inhaber.

Mit Maschinencode wird in dieser Arbeit ausführbarer Binärcode für physikalisch existierende Maschinen bezeichnet. Zur besseren Unterscheidung wird von ausführbarem Code für die virtuelle Java Maschine stets als Bytecode gesprochen.

Tiling bezeichnet die Zerlegung eines Problems in mehrere Teilprobleme. Üblicherweise ist die Anzahl der Teilprobleme gleich der Anzahl der zur Berechnung herangezogenen Prozesse. Da letztere in der Regel kleiner ist als die Problemgröße, entfallen auf jeden Prozeß mehrere Elemente (ein Tile), die dann zumeist sequenziell berechnet werden.

Einige Begriffe aus der objektorientierten Programmierung werden im Text als bekannt vorausgesetzt. Dazu gehören u.a.:

Objektvariablen: Oberbegriff für Variablen, die auf oberster Ebene[8] innerhalb einer bestimmten Klasse deklariert wurden. Die Klasse `A`:

```
public class A {
  int b;
}
```

enthält genau eine Objektvariable `b` von Typ `int`. Je nach Sichtbarkeit wird zwischen Instanz- und Klassenvariablen unterschieden.

[6] Die Zielplattformen der Projekte sind zum Teil sehr verschieden (JVM auf SHM-Multiprozessor / Maschinencode-Implementierung für NOW); zudem stand nicht für jede Sprache die optimale Plattform zum Test zur Verfügung.

[7] Die Unterschiede zwischen diesen Versionen sind für den Inhalt dieser Arbeit ohne Bedeutung. Die untersuchten Projekte beziehen sich jedoch zumeist auf die Version 1.1.

[8] Die Oberste Ebene ist dabei die Ebene, auf der auch Methoden der betrachteten Klasse deklariert würden. Lokale Variablen innerhalb einer Methode sind somit keine Objektvariablen.

Instanzvariablen: Jedes Objekt besitzt eine eigene Instanz dieser Variablen. Die Variable `A.b` ist eine Instanzvariable.

Klassenvariablen: Objektvariablen, welche als `static` deklariert sind. Alle Objekte einer Klasse teilen sich eine Instanz der Variable.

`static` Variablen: Synonym für Klassenvariablen.

Hinzu kommen eine Reihe von Begriffen, die sehr javaspezifisch sind. Diese werden in Kapitel 2 genauer erklärt. Viele Fachbegriffe in dieser Arbeit wurden vom Englischen ins Deutsche übersetzt.

2. Java

In diesem Kapitel werden einige Details des Java-Standards dargestellt, die für das Verständnis der nachfolgenden Kapitel hilfreich sind.

2.1. Maschinenmodell

Java-Programme werden nicht in "echten" Maschinencode, d.h. Anweisungen für eine real existierende Maschine übersetzt, sondern in sogenannten Bytecode. Dieser Bytecode (oder JVM-Maschinencode) wird von einer virtuellen Maschine (JVM oder VM) interpretiert. Eine (Software-) Implementierung dieser VM existiert mittlerweile für die meisten Plattformen (inklusive Chipkarten und Organizer) und macht Java-Programme auf diesen Plattformen ausführbar. Obwohl sich Java-Bytecode relativ effizient interpretieren läßt, bringt die zusätzliche Softwareschicht gewisse Geschwindigkeitsnachteile mit sich.

Um die Laufzeit von Java-Programmen zu verbessern, verfügen die meisten virtuellen Maschinen mittlerweile über sogenannte Just-In-Time-Compiler (JIT-Compiler)[1]. Diese übersetzen den Bytecode zur Laufzeit in Maschinencode für die jeweilige Zielarchitektur. Die so erreichte Leistung ist wesentlich besser als die von rein interpretiertem Bytecode, bleibt jedoch weiterhin hinter der von (z.B.) C/C++ Programmen zurück[2]. Der Grund hierfür liegt an zahlreichen Laufzeittests (Überschreiten von Indexgrenzen, Verwendung von `null`-Zeigern, unerlaubte Typkonvertierungen, etc.), die selbst nach einer Übersetzung in Maschinencode Mehraufwand gegenüber konventionellen C/C++-Programmen bedeuten. Abhilfe könnten hier optimierende Compiler schaffen, die unnötige Laufzeittests eliminieren [6].

2.2. Objekte und Referenzen

Ein herausragendes Merkmal von Java ist seine konsequente Objektorientierung; Zeichenketten werden ebenso als Objekte (`java.lang.String`) repräsentiert, wie Arrays oder Threads (`java.lang.Thread`, vgl. Abschnitt 2.6). Jedes Objekt ist dabei implizit von `java.lang.Object` abgeleitet, ohne daß dies einer entsprechenden Deklaration durch den Programmierer bedarf. Entgegen einer verbreiteten Meinung gibt es auch in Java Zeiger, wenngleich sich Ihre Verwendung von der in C/C++ unterscheidet.

[1] Im Verlauf dieser Arbeit wird zudem des öfteren (insbesondere in Diagrammen) die Abkürzung "*nojit*" für "ohne Just-In-Time-Compiler" benutzt.

[2] Ein Vergleich der Laufzeiten von Java und C/C++ Programmen folgt in Abschnitt 7.4.2.

- Es gibt keine Zeiger auf einfache (skalare) Werte.
- Es gibt keine "objektwertigen" Variablen; eine Variable vom Typ "Objekt" ist implizit ein "Zeiger-auf-Objekt".
- Bei Methodenaufrufen werden skalare Werte stets per Kopie, Objekte stets per Referenz übergeben.
- Der Zugriff auf skalare Variablen erfolgt stets direkt, der auf Objekte stets per Referenz. Einen Referenzoperator (wie `*` oder `&` in C/C++) gibt es nicht.

Abgesehen von acht skalaren Typen (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double`) und `void`, sind alle Datentypen in Java objektorientiert. Da alle Objekte in Java implizit von `java.lang.Object` abgeleitet sind, ist eine Variable vom Typ `java.lang.Object` somit ein gültiger "Universalzeiger". Dieser Umstand wird z.B. von den meisten Containerklassen der Java-Standardbibliotheken benutzt, um beliebige Objekte in dynamischen Strukturen abzulegen.

Ein detailliertes Laufzeittypsystem verhindert unerlaubte Typkonvertierungen. Zudem ist es zur Laufzeit möglich, detaillierte Informationen über ein Objekts zu erfragen. Das hierfür benutzte "Reflection"-Paket bietet unter anderem Zugriff auf.

- Den Typ (die Klasse) des Objekts, inklusive Informationen über die Basisklasse und eventuell implementierte Schnittstellen[3].
- Die Namen und Typen aller Objektvariablen, sowie deren Zugriffsbeschränkungen (`public`, `private`, ...).
- Die Namen und Signaturen aller Methoden; dazu gehören ebenfalls Zugriffsbeschränkungen sowie von dieser Methode eventuell verursachte Ausnahmen.
- Die gleichen Informationen für alle inneren Klassen.

2.3. Vererbung

Anders als z.B. in C++ gibt es in Java keine Mehrfachvererbung. Jede Klasse besitzt höchstens eine Basisklasse. Zusätzlich zur Vererbung kann eine Klasse allerdings beliebig viele *Interfaces* implementieren. Ein `interface` enthält nur Variablen- und Methoden-Deklarationen, jedoch keine Implementierung der Methoden. Diese implementierten Schnittstellen sind (genauso wie die Basisklassen) Teil der Typinformation. Eine Klasse `ComplexNumber`:

```
class ComplexNumber extends Number implements Printable, Editable {
  // Implementation ...
```

[3] Die Details zur Vererbung in Java werden in Abschnitt 2.3 erläutert.

ist somit vom Typ `java.lang.Object`, `java.lang.Number`, `Printable` und `Editable`. Eine Konvertierung zwischen diesen Typen ist zur Laufzeit möglich. Wird eine Methode von mehr als einem benutzen Interface deklariert, so überprüft der Compiler ob die Signaturen identisch sind. Existieren zwei gleichnamige Deklarationen mit identischen formalen Parametern aber unterschiedlichem Rückgabewert, so wird beim Übersetzen eine entsprechende Fehlermeldung ausgegeben.

Java erlaubt seit der Version 1.1 die Definition sogenannter *innerer Klassen*. Diese Klassen werden innerhalb einer anderen Klasse deklariert und bieten eine interessante (und sichere) Alternative zur Mehrfachvererbung.

```
class OuterClass {
  // Implementation ...

  class InnerClass extends SomeClass{
    // Implementation ...
  }

}
```

In diesem Beispiel erbt die innere Klasse `InnerClass` die Variablen und Methoden ihrer Basisklasse `SomeClass`, hat aber gleichzeitig Zugriff auf die Elemente der umgebenden Klasse `OuterClass`.

2.4. Skalare Typen

Um auch skalare Variablen wie Objekte behandeln zu können, gibt es für jeden einfachen Typ (`int`, `long`, `double`...) eine entsprechende Klasse (`java.lang.Integer`, `java.lang.Long`, `java.lang.Double`...) mittels der sich geeignete *Wrapper* erzeugen lassen.

```
double theValue = 42.0;
Double theWrapper = new Double(theValue);
```

Diese Umschläge werden immer dann benötigt, wenn skalare Typen als Objekte übergeben werden sollen. So erwarten z.B. die meisten der dynamischen Datenstrukturen aus den Java Bibliotheken Argumente vom Typ `java.lang.Object`.

Leider gibt es in Java keine einfache Möglichkeit, direkt auf den objektorientierten Typen zu rechnen[4]; eine implizite Konvertierung zwischen einfachen Typen und ihren Umschlägen erfolgt nur in (wenigen) bestimmten Fällen. Zudem ist die Verwendung dieser Umschläge oft mit deutlichen Geschwindigkeitsnachteilen verbunden. Wie [25] feststellt, ist auf der dort benutzten Referenzplattform die Erzeugung eines `java.lang.Double`-Objektes mehr als 175 mal langsamer, als die Erzeugung einer `double` Variable.

[4] Abschnitt 3.4.2 zeigt, wie auch mit objektorientierten numerischen Typen gerechnet werden kann. Allerdings ist diese Methode vergleichsweise ineffizient.

2.5. Arrays

Arrays werden in Java als Objekte repräsentiert. Dabei wird die Arraygröße erst bei der Instanziierung festgelegt, ist also nicht (wie z.B. in Pascal) Teil der Typinformation. Da Arrayelemente (ebenso wie alle anderen Variablen in Java) bei der Instanziierung des Arrays initialisiert werden[5], ist die hierfür benötigte Zeit linear in der Größe des Arrays [25].

Mehrdimensionale Arrays wie z.B.

```
int[][] a = new int[42][42]
```

werden in Java durch Arrays von Arrays repräsentiert, wodurch sich eine Priorisierung der Dimensionen ergibt. Das bedeutet z.B., daß der Zugriff auf einzelne Zeilen eines zweidimensionalen Arrays wesentlich effizienter sein kann, als der Zugriff auf einzelne Spalten. In einem Ausdruck wie

```
a[i][j]
```

bestimmt der am weitesten rechts stehende Index (in diesem Fall also `j`), das Element in der Zeile, während `i` die Zeile bestimmt. Da in Java die Zeilen eines Arrays unterschiedliche Länge haben dürfen, sollte dies bei der Behandlung von Indexgrenzen beachtet werden.

Eine einfache Möglichkeit, ein mehrdimensionales Array als ein eindimensionales Array geeigneter Größe zu behandeln gibt es nicht. Ebensowenig ist es möglich, beliebige Subarrays zu referenzieren, oder die Prioritäten der Dimensionen zu vertauschen.

2.6. Multithreading

Das Threadmodell in Java ist dynamisch. Jeder Thread wird durch ein `java.lang.Thread`-Objekt repräsentiert.Dieses Objekt bietet zum einen Zugriff auf den Zustand des Threads (ausführbar, beendet), zum anderen ist die Aktionsfunktion des Threads durch das Threadobjekt festgelegt. Der Programmierer hat dabei die Wahl, entweder die Methode `run` der Klasse Thread in einer abgeleiteten Klasse geeignet zu überladen, oder bei der Erzeugung des Threads dem Konstruktor eine Referenz auf ein `Runnable`-Objekt zu übergeben. `Runnable` ist dabei eine einfache Schnittstelle, in der lediglich die Signatur der `run`-Methode festgelegt ist. Eine Möglichkeit, beliebige Methoden direkt durch einen Thread ausführen zu lassen gibt es nicht.

Die übliche Methode zum Erzeugen neuer Threads in einem Java-Programm besteht darin, ein Objekt vom Typ `Thread` zu instantiieren und anschließend dessen Methode `start` aufzurufen. Dadurch wird der Thread in den Zustand "ausführbar" versetzt und die Laufzeitumgebung sorgt dafür, daß zu einem späteren Zeitpunkt die Methode `run` abgearbeitet wird. Viele Java-Implementierungen verwenden ein nicht preemptives Scheduling; um die CPU dennoch jederzeit explizit abzugeben, kann der Methode

[5] Die Initialisierung erfolgt mit geeigneten Nullwerten (also `null`, 0 oder 0.0); neue Objekte werden hierbei nicht erzeugt.

`Thread.yield` benutzt werden. Diese sorgt dafür, daß die Kontrolle an einen beliebigen anderen rechenwilligen Thread übergeht.

Abgesehen davon, daß die objektbasierte Thread-Implementierung Leistungsnachteile mit sich bringt, gibt es einige Besonderheiten, die zu beachten sind:

- Seit der Version 1.2 gibt es keine Möglichkeit mehr, einen laufenden Thread vorzeitig zu beenden. Die hierfür vorgesehene Methode `stop` soll (aus berechtigten Sicherheitsüberlegungen) nicht mehr benutzt werden[6] und wird in späteren Versionen nicht mehr implementiert sein. Dies führt allerdings dazu, daß jeder Thread nur dann terminiert, wenn seine `run`-Methode beendet wird (entweder regulär oder durch das Auftreten einer Ausnahme), was bei (z.B. auf externe Ereignisse wie IO) wartenden Threads nicht immer der Fall ist.

- Über die Methode `interrupt` läßt sich an der aktuellen Ausführungsposition des entsprechenden Threads eine `java.lang.InterruptedException` erzeugen, sofern der Thread gerade wartend gesetzt ist. Andernfalls wird lediglich eine Marke gesetzt, deren Zustand explizit über die Methode `isInterrupted` abgefragt werden muß. Abbildung 2.1 zeigt einen Thread, der von einem `Thread.interrupt` nicht beendet werden kann.

```
class DifficultToStop extends Thread {

    public void run() {
      synchronized {

        while (true) {
          try {
            wait();
            return; // return only if notified.
          } catch ( InterruptedException e ) {
            // ignore exception...
          }
        }

      }
    }

}
```

Abbildung 2.1.: Dieser Thread wird durch `Thread.interrupt` nicht beendet

[6] Die Methode `destroy`, die einen ähnlichen Effekt haben sollte, wurde ohnehin nie implementiert.

2.6.1. Monitore in Java

Ein Monitor ist ein abstrakter Datentyp der den konfliktfreien Zugriff mehrerer Threads auf eine gemeinsame (in der Regel unteilbare) Ressource ermöglichen soll. Die Idee hierfür wurde 1974 von Hoare entwickelt. In der Modellvorstellung ist der Monitor ein Raum, in dem sich zu jedem Zeitpunkt maximal ein Thread aufhalten kann. Operationen mit der geschützten Ressource sind nur innerhalb des Monitors möglich.

Um Operationen innerhalb des Monitors ausführen zu können muß der entsprechende Thread zunächst Zutritt zum Monitor erlangen. Hält sich zum Zeitpunkt des Zutrittswunsches kein anderer Thread innerhalb des Monitors auf, so wird der Zutritt sofort gewährt. Andernfalls wird der anfordernde Thread wartend gesetzt und der Warteliste für diesen Monitor hinzugefügt. Verläßt der Thread im inneren des Monitors diesen, so wird der jeweils nächste Thread aus der Warteliste geweckt und darf den Monitor betreten.

Zusätzlich zu den Wartelisten für den Zutritt, besitzt jeder Monitor eine oder mehrere sogenannte *Bedingungswarteschlangen*. Diese erlauben es Threads, den Monitor bis zum Eintreten einer Bedingung (z.B. das Füllen eines gemeinsam benutzten Puffers durch einen anderen Thread) freizugeben. Um dies zu erreichen ruft der wartewillige Thread die Methode `cond.wait` der entsprechenden Warteschlange auf, was dazu führt, daß der Thread wartend gesetzt, und der entsprechenden Bedingungswarteschlange hinzugefügt wird. Gleichzeitig wird ihm der Monitor entzogen um anderen Threads den Zutritt zu ermöglichen.

Das Wahrwerden einer Bedingung (also etwa das Füllen des Puffers) kann von dem dafür verantwortlichen Thread signalisiert werden, indem er die Methode `cond.signal` der entsprechenden Warteschlange aufruft. Daraufhin wird der Thread mit der höchsten Priorität aus der entsprechenden Warteschlange entfernt und kann seine Aktionen innerhalb des Monitors fortsetzen. Für die genaue Implementierung des Signalmechanismus gibt es verschiedene Möglichkeiten, die in [4, 15] näher untersucht werden:

Automatic Signaling (AS): Das Wecken eines wartenden Threads erfolgt implizit beim Wahrwerden einer Bedingung. Diese Variante existiert nur in Verbindung mit einem `await`-Konstrukt und sei hier nur der Vollständigkeit halber genannt.

Signal and Continue (SC): Das Wecken eines wartenden Threads erfolgt explizit durch den Aufruf von `cond.signal`, dieser muß sich dann (in Konkurrenz zu anderen Threads) erneut um den Monitor bewerben. Der aktiver Thread behält den Monitor nach dem Aufruf von `cond.signal` solange, bis er diesen (freiwillig) verläßt.

Signal and Exit (SE): Das Wecken eines wartenden Threads erfolgt explizit durch den Aufruf von `cond.signal`, ihm wird der Monitor sofort zugeteilt und der aktive Thread verläßt mit dem `cond.signal`-Aufruf den Monitor.

Signal and Wait (SW): Das Wecken eines wartenden Threads erfolgt explizit durch den Aufruf von `cond.signal`, ihm wird der Monitor sofort zugeteilt und der aktive Thread wird der Warteschlange für diesen Monitor hinzugefügt.

Signal and Urgent Wait (SU): Das Wecken eines wartenden Threads erfolgt explizit durch den Aufruf von `cond.signal`, ihm wird der Monitor sofort zugeteilt, der aktive Thread wird der Warteschlange für diesen Monitor hinzugefügt, darf aber den Monitor vor allen anderen wieder betreten.

Warum man sich bei der Java-Spezifikation auf SC festgelegt hat bleibt unklar, zumal von Hoare selbst die Methode SU empfohlen wird. Untersuchungen in denen die Effizienz der Verfahren im Vordergrund steht bestätigen Hoare's Aussage und bescheinigen SU zudem bessere Kontrolle über die wartenden Prozesse sowie die Reihenfolge der Aktivierung[15]. Lediglich die geringen Implementierungskosten und die einfache Benutzung sprechen nach [4] für SC, was zumindest im Bereich des Hochleistungsrechnens jedoch von untergeordneter Bedeutung sein dürfte.

In Java ist mit jedem Objekt stets (genau) ein Monitor und eine Wartemenge verbunden. Die Wartemenge entspricht dabei einer Bedingungswarteschlange, wie sie in diesem Abschnitt vorgestellt wurde. Für den Zugriff auf diese Wartemenge definiert jedes Java-Objekt die Methoden `wait` und `notify`, die den bereits vorgestellten Methoden `cond.wait` und `cond.signal` entsprechen. Bevor ein Thread eine dieser Methoden ausführen darf, muß er sich erfolgreich um den (mit dem Objekt assoziierten) Monitor bewerben.

Anders als in Bedingungswarteschlangen, existiert in der Wartemenge eines Objekts keine Ordnung unter den wartenden Threads. Der Aufruf von `notify` weckt somit einen beliebigen, und nicht (z.B.) den am längsten wartenden Thread. Zusätzlich existiert für Java-Objekte die Methode `notifyAll`, die alle Threads in der Wartemenge aufweckt und zu der Menge der um den Monitor konkurrierenden Threads hinzufügt.

Da Monitore in Java keine sprachliche Entsprechung haben (es gibt kein Objekt von Typ "Monitor"), werden sie stets über die Objekte referenziert, mit denen Sie verbunden sind. Diese Praxis wird zum Teil auch in dieser Arbeit nachvollzogen indem am weiteren kurz von Monitorobjekten, anstelle von "mit den Objekten assoziierten Monitoren" gesprochen wird.

2.6.2. Synchronisation von Java Threads

Für den gegenseitigen Ausschluß über Monitore wird in Java das Schlüsselwort `synchronized` benutzt. Ein als `synchronized` markierter Block wird zu jedem Zeitpunkt von höchstens einem Prozeß ausgeführt. Verschiedene Schreibweisen erlauben die Auswahl welcher Monitor zur Synchronisation benutzt werden soll; dazu wird wie bereits erwähnt das Java-Objekt angegeben, dessen Monitor benutzt werden soll. Abbildung 2.2 zeigt die am häufigsten benutzte Verwendung dieses Schlüsselworts. Die Methode `meAlone` ist als `synchronized` markiert, die Synchronisation erfolgt über den Monitor des Objekts zu dem auch die Methode gehört. Während die Methode `anyoneElse` jederzeit von beliebig vielen Prozessen ausgeführt werden darf, kann `meAlone` stets von höchstens einem Prozeß ausgeführt werden.

```
class Monitor {

    public synchronized void meAlone() {
        // synchronized code...
    }

    public void anyoneElse() {
        // regular code...
    }
}
```

Abbildung 2.2.: Synchronisierte Methoden in Java

Der gegenseitige Ausschluß bezieht sich nur auf synchronisierte Blöcke, die denselben Monitor benutzen. Abbildung 2.3 zeigt, wie dieser Umstand benutzt werden kann, um mehreren Prozessen den Zutritt zu einem synchronisierten Block zu erlauben. Die Methode `meAndMyFriends` enthält einen als `synchronized` markierten Block, für den das zu benutzende Monitorobjekt explizit angegeben wurde. Ein Pool von Objekten (`lock[]`) stellt `MAX_PROC` verschiedene Monitorobjekte zur Verfügung. Anzumerken ist, daß diese Implementierung ist nicht fair ist, da die Methode `requestLock` die Monitorobjekte unabhängig davon zuteilt, ob der betreffende Monitor frei ist oder nicht. Die Auswahl wartender Prozesse erfolgt in Java nichtdeterministisch, wodurch einzelne Prozesse verhungern können.

2.7. Speichermodell

Der zur Verfügung stehende Speicher wird in Java ausschließlich von der virtuellen Maschine verwaltet. Die Sprache kennt keine Operationen auf Zeigern und Adressen (wie C oder C++). Selbst auf JVM-Assemblerebene gibt es keine Möglichkeit, Speicher explizit zu belegen oder freizugeben. Dies bringt zum einen Vorteile im Hinblick auf Sicherheit und Stabilität, erfordert zum anderen aber das automatische Finden und Freigeben von nicht mehr benötigten Objekten durch die VM. Dieser Speicherbereinigung (oder engl. *Garbage Collection*) genannte Vorgang läuft asynchron zur eigentlichen Programmausführung und ist für verteilte Programme mit ungesicherter Verbindungen zwischen den einzelnen Rechnern (Internet-Computing) unter Umständen schwer zu realisieren. Da die Speicherbereinigung nur angestoßen wird, wenn der verfügbare Speicher knapp wird, lassen sich bei großen Problemgrößen in verteilten Programmen deutliche Beschleunigungen beobachten (wenn Aufgrund der Aufteilung die Speicherbereinigung ausbleibt oder deutlich seltener stattfindet).

```
int i=0;
Object lock[] = new Object[MAX_PROC];
... create instances and assign to lock[] elements.


// synchronized to avoid race conditions:

public synchronized Object requestLock() {
  i = (i+1)%MAX_PROC;
  return lock[i];
}

// allows up to MAX_PROC processes to
// enter the synchronized block:

public void meAndMyFriends() {
  Object myLock = requestLock();

  synchronized (myLock) {
    i = (i+1)%MAX_PROC;
      // in monitor....
  }
}
```

Abbildung 2.3.: Synchronisation über benannte Monitore

Der Speicher innerhalb einer Java VM steht allen Threads gemeinsam zur Verfügung. In diesem Zusammenhang stellt sich die Frage, wie konkurrierende Zugriffe verschiedener Threads auf diesen Speicher organisiert werden. Java verwendet ein schwaches Konsistenzmodell für Zugriffe auf gemeinsame Variablen. In [18, S. 401] heißt es dazu: "... a Java implementation is free to update the main memory in an order that may be surprising[7]." Konkret bedeutet dies, daß verschiedene Threads innerhalb einer virtuellen Maschine zeitweise verschiedene lokale Kopien einer Variablen besitzen dürfen. Ein Zurückschreiben der geänderten Werte ist nur an bestimmten Stellen vorgeschrieben, unter anderem beim Betreten und Verlassen von als `synchronized` markierten Blöcken. Eine Ausnahme bilden Variablen, die mit dem Modifier `volatile` deklariert wurden, denn dieser verbietet die Erzeugung lokaler Kopien einer Variablen.

Leider lassen sich zwar Array-Variablen (also Variablen vom Typ "Array von-...") als `volatile` deklarieren, jedoch nicht die Elemente eines Arrays. In [10] werden Arrays von `boolean` Variablen als Bitvektoren zur Synchronisation benutzt. Da es nicht möglich ist, jedes einzelne Bit dieser Vektoren als `volatile` zu deklarieren, kann dies (wegen sich un-

[7]Die Quelle benutzt tatsächlich das Wort "surprising", weshalb dieser Halbsatz zitierenswert erschien.

terscheidender lokaler Kopien) in Spezialfällen zu Inkonsistenzen führen. Da die Zugriffe auf solche Vektoren in der Regel nicht direkt, sondern über geeignete (synchronisierte) Methodenaufrufe erfolgen (nicht zuletzt, weil dies in Java die einzige Möglichkeit ist, ein "Test-and-Set" als unteilbare Operation zu realisieren), dürften sich die Auswirkungen dieser Besonderheit in Grenzen halten, wie auch die in Kapitel 7 durchgeführten Tests bestätigen[8].

2.8. Netzwerkprogrammierung

Im Rahmen der Standardbibliotheken stellt Java Klassen für den einfachen und plattformunabhängigen Zugriff auf verschiedene Netzwerkfunktionen zur Verfügung. Eine Socketverbindung zu Port `1234` auf dem Rechner `otherhost` läßt sich einfach mittels

```
Socket s = new Socket(‘‘otherhost’’,1234);
```

herstellen. Das Socketobjekt `s` besitzt Methoden `getInputStream` und `getOutputStream`, über die auf die Netzwerkverbindung in der gleichen Weise zugegriffen werden kann, wie auf eine geöffnete Datei. Die Klasse `DataOutputStream` und ihr Gegenstück `DataInputStream` stellt Methoden bereit, um die wichtigsten Datentypen in einen `OutputStream` zu schreiben, bzw. aus einem `InputStream` zu lesen.

```
DataOutputStream out = new DataOutputStream(s.getOutputStream());

out.writeDouble(1.2345678);
out.writeLong(42);
...
```

Durch den Einsatz dieser Klassen ist die Kommunikation zwischen verschiedenen Plattformen möglich, ohne daß der Programmierer Eigenheiten wie unterschiedliche Byte-Ordnungen oder Wortbreiten beachten muß. So lassen sich in verteilten Anwendungen kostengünstige Kommunikationskanäle realisieren, was in einigen der Fallstudien in Kapitel 7 auch demonstriert wird.

Zusätzlich gibt es eine Reihe weiterer Klassen etwa für Namensauflösung, Multicast-Kommunikation oder verschiedene Server-Dienste. Eine genaue Erläuterung würde hier jedoch zu weit führen.

2.9. Remote Method Invocation (RMI)

Java bietet eine Möglichkeit zur Kommunikation mit virtuellen Maschinen auf entfernten Rechnern, die auf Sockets basiert. RMI ist das objektorientierte Gegenstück zum RPC

[8] In keinem der Beispiele kam es zu Problemen, die sich auf den beschriebenen Effekt zurückführen ließen.

(Remote Procedure Call) und bietet eine komfortable Möglichkeit, Methoden auf entfernten Rechnern ausführen zu lassen. Um dies zu ermöglichen werden sogenannte *stub* und *skeleton* Klassen generiert, die den netzwerkspezifischen Anteil eines entfernten Methodenaufrufes vor dem Benutzer verbergen. Aus Programmierersicht können Methoden eines entfernten Objektes (beinahe) auf die gleiche Art aufgerufen werden, wie die eines lokalen Objektes.

Der Zugriff auf Methoden von entfernten Objekten erfolgt über Schnittstellen, die sowohl von dem eigentlichen Objekt, als auch von den erzeugten Stubobjekten implementiert werden. Wird eine Methode des Stubobjektes aufgerufen, so wird der Aufruf über eine Netzwerkverbindung an das entsprechende Skeletonobjekt und von diesem schließlich an das eigentliche Objekt weitergeleitet. Dazu werden alle Parameter (wenn nötige der gesammte von dem zu übergebenden Objekt ausgehende Objektgraph) in einem *Marshalling* (oder oft auch Serialisierung) genannten Verfahren in eine serielle Darstellung umgewandelt, übertragen und von dem Skeletonobjekt wieder in Objekte umgesetzt. Auf diese Art entsteht auf der gegenüberliegenden Seite eine identische Kopie jedes Parameters. Skalare Typen lassen sich mit diesem Verfahren ebenfalls übertragen; sie werden zum Transfer in geeignete Umschlagobjekte verpackt (vgl. Abschnitt 2.4). Zusätzliche Überprüfungen stellen sicher, daß jedes Objekt nur einmal übertragen wird und somit die referentielle Integrität zumindest innerhalb eines Aufrufes erhalten bleibt. Einige Besonderheiten sind zu beachten:

- Die Parameterübergabe an entfernte Methoden erfolgt im allgemeinen per Wert und nicht per Referenz wie normalerweise für Objekte in Java üblich.

- Referentielle Integrität für übergebene Parameter gilt nur innerhalb eines Aufrufes. Während für den Aufruf:

  ```
  Object b = new Object;
  remoteObject.remoteMethod(b,b);
  ```

  nur ein Objekt `b` (und eine Referenz darauf) übergeben wird, werden für die Aufrufe:

  ```
  Object b = new Object;
  remoteObject.remoteMethod(b);
  remoteObject.remoteMethod(b);
  ```

  tatsächlich zwei verschiedene Objekte übergeben.

- Das für die Übergabe von Objekten per Wert benutzte Marshalling / Unmarshalling Verfahren verlangt, daß alle zu übergebenden Objekte die Schnittstelle `Serializable` implementieren, was leider nicht bei allen Klassen der Standardbibliotheken der Fall ist. Wird ein Objekt übergeben, so werden alle davon referenzierten Objekte (also die Hülle dieses Objekts in einem gewissem Sinne) ebenfalls kopiert. Eine Ausnahme bilden Objektvariablen, die als `transient` deklariert wurden. Diese sind per Definition von der Serialisierung ausgenommen. Die Standardimplementierung der Objektserialisierung war ursprünglich für die persistente Speicherung in Dateien konzipiert und ist daher in einigen Implementierungen nicht besonders effizient[9].

Während eines RMI-Methodenaufrufs ist der aufrufende Thread blockiert. Dies entspricht der erwarteten Semantik eines Methodenaufrufes: die Aktionsfunktion wird solange in dem aufgerufenen Codeblock (und somit auf der JVM, die diesen Codeblock "besitzt") fortgesetzt, bis dieser wieder verlassen wird. Um zusätzlich zur (so erzielten) Verteilung auch echte Parallelität zu erreichen, muß es auf der aufrufenden Maschine wenigstens einen Thread geben, der während des RMI-Aufrufes rechenwillig ist, und somit parallel dazu Aktionen ausführen kann. Nur in diesem Fall findet tatsächlich eine Parallelverarbeitung auf beiden beteiligten Maschinen statt.

Derzeit wird eine Integration von RMI und CORBA entwickelt, die die Vorteile beider Techniken kombinieren soll[10]. Das zugrundeliegende RMI-IIOP Protokoll ermöglicht den Aufruf von CORBA-Objekten aus Java-Programmen und bringt eine Reihe von interessanten Anwendungsmöglichkeiten mit sich. Beide Ansätze sind jedoch eher auf verteilte Anwendungen im Sinne von Dienstanbietern und -nehmern ausgerichtet und eignen sich vor allem für grobkörnige Parallelität.

2.10. Ausnahmen

Auf Ebene der virtuellen Maschine gibt es drei verschiedene Arten von Ausnahmen:

Virtual-Machine-Exceptions: können bei jeder JVM-Anweisung auftreten und signalisieren einen internen Fehlerzustand oder Resourcenmangel, der die weitere Ausführung unmöglich macht.

Linking-Exceptions: können bei u.a. jeder Instanziierung eines neuen Typs und bei jedem Methodenaufruf auftreten, da Java-Programme generell zur Laufzeit gebunden werden.

Runtime-Exceptions: können entweder explizit durch eine Programmanweisung oder implizit z.B. beim Überschreiten von Array-Indexgrenzen auftreten.

Während Virtual-Machine-Exceptions an jeder beliebigen Stelle im Programm auftreten können, ist in der JVM-Spezifikation [28] genau festgelegt, welche Anweisungen Runtime- oder Linking-Exceptions verursachen können.

Das Exceptionmodell von Java verursacht zahlreiche Probleme bei der Optimierung, wie folgendes Beispiel aus [12] zeigt:

[9] In [29] finden sich genauere Untersuchungen zu RMI sowie konkrete Vorschläge zu Optimierungen. Ein effizienteres Verfahren zur Objekt-Serialisierung wird in [31] vorgestellt.

[10] Details hierzu finden sich unter: `http://java.sun.com/products/rmi-iiop/`, bzw. `http://java.sun.com/products/jdk/1.3/docs/guide/rmi-iiop/`

```
for (i=k; a[i] != a[i-k] ; i++) {
  this.x = k;
  a[i] = a[i-k];
}
```

Da `k` schleifen-invariant ist, ist auch der Ausdruck `this.x = k` schleifen-invariant. Die übliche Optimierungsmethode wäre, den entsprechenden Code aus dem Schleifenrumpf nach außen zu verschieben. Um die Exception-Semantik zu erhalten, darf jedoch kein Code über eine Anweisung hinweg verschoben werden, die eine Ausnahme verursachen kann. Da in Java bei jedem Array-Zugriff eine `ArrayIndexOutOfBounds`-Exception auftreten kann, verhindert der Array-Zugriff im Kopf der Schleife zunächst diese Optimierung.

Um zu garantieren, daß Ausnahmen (obwohl theoretisch möglich) nicht auftreten werden, sind zumeist Tests zur Laufzeit erforderlich. So läßt sich die Schleife aus obigem Beispiel wie folgt umschreiben:

```
if ( (k>=0) && (k<a.length) && (a[k] != a[0]) ) {
  this.x = k;
}
for (i=k; (i>=0) && (i<a.length) && ((i-k)>=0) && ((i-k)<a.length)
      && (a[i] != a[i-k]) ; i++) {
  a[i] = a[i-k];
}
```

Die so modifizierte Schleife wird regulär beendet, selbst wenn eine Verletzung der Array-Grenzen auftritt und erlaubt eine Verschiebung des invarianten Ausdrucks `this.x = k` aus der Schleife. Eine genauere Untersuchung solcher Transformationen findet sich in [6]

3. Titanium

Titanium wird an der University of California at Berkeley entwickelt. Ziel des Projekts ist die Entwicklung einer Sprache für verschiedene Parallelarchiktekturen. Java wurde als Ausgangspunkt gewählt, wobei versucht wurde, eine Obermenge des Sprachumfangs zu definieren, die die Kompatibilität zu "gewöhnlichen" Java-Programmen erhält. `tcbuild`, der Titanium-Compiler besitzt sogenannte *Backends* für verschiedene Kommunikationsbibliotheken (genaueres weiter unten); die Schwerpunkte bei der Entwicklung waren (mit abnehmender Gewichtung): Performance, Sicherheit und Ausdruckskraft [36].

Die neu entwickelte Sprache soll die Implementierung von wissenschaftlichen Anwendungen unterstützen.,Dabei wurde die Kompatibilität zum allgemeinen Java-Standard zwar angestrebt, jedoch wurde sie nie als zwingend notwendig erachtet. Dadurch ist Titanium keine "echte" Obermenge von Java; bewußt ausgeklammert wurde etwa das dynamische Threadmodell (genaueres in Abschnitt 3.2.1). Die javaähnliche Syntax und die weitreichende Kompatibilität sind beim Erlernen der Sprache, sowie bei der Portierung bestehender Applikationen dennoch hilfreich. Die wichtigsten Bibliotheken des Java-Standards, `java.lang`, `java.io` und `java.util`, stehen in Titanium als *tlib* Bibliothek zur Verfügung, wodurch zumindest für einfache Programme kein großer Unterschied zu Java besteht. Obwohl sich Titanium laut Dokumentation am Java 1.1 Standard orientiert, sind leider (ebenso wie in Java 1.0) keine inneren Klassen erlaubt[1].

Als klassischer Vertreter des *Dialekt*-Ansatzes (vgl. Abschnitt 1) ist die Zielsprache maschinenabhängig; die aktuelle Version benutzt C++ als Zwischensprache. Besonderes Augenmerk wurde bei der Entwicklung darauf gelegt, daß der erzeugte Zwischencode keine großen Anforderungen an einen optimierenden C-Compiler stellt. Da Art und Umfang der Optimierungen von Compiler zu Compiler variieren und zudem von der Art der Quellprogramme und der Zielarchitektur abhängig sind, ist es nicht immer einfach, dieses Ziel zu erreichen. Titanium führt unter anderem folgende Analysen und Optimierungen durch:

- Finden von Definitionen und Zugriffen auf Variablen.
- Berechnung aller möglichen Ausführungspfade.
- Finden und Verschieben von schleifen-invarianten Ausdrücken.
- *Strength reduction* von Array-Index-Ausdrücken.

[1] Da es in Java keine Mehrfachvererbung gibt, sind innere Klassen ein wegen ihrer Ausdrucksstärke häufig benutztes Sprachkonstrukt (vgl. Abschnitt 2.3).

- Unterdrückung einer Kontroll-Variablen für `foreach` Ausdrücke, falls diese nicht benötigt wird.

Der wichtigste Vorteil gegenüber dem Bibliotheksansatz (vgl. Abschnitt 1) ist sicherlich die erreichbare Ausführungsgeschwindigkeit[2]. Zielprogramme in Maschinencode werden (zumindest auf absehbare Zeit) denen in interpretiertem Bytecode überlegen sein. Titanium bietet die Möglichkeit, Code für verschiedene Zielarchitekturen zu erzeugen, dadurch sind Titanium-Programme sowohl auf Einprozessor- als auch auf SMP-Architekturen und Rechnernetzen lauffähig. Derzeit werden folgende Backends unterstützt:

uniproc: Ist immer verfügbar und erzeugt rein sequenziellen Code.

pthread: Multithreading mit Hilfe der POSIX Thread-Bibliothek (*libpthread*), die u.a. für Solaris und Linux verfügbar ist.

smp-narrow: Geeignet für Solaris SMP Workstations mit der entsprechenden Thread-Bibliothek (*libthread*).

split-c: Unterstützung für ein Arbeitsplatznetz (Network of Workstations, kurz: NOW) mit Hilfe der Split-C Bibliothek (lisplit-c) und der Active-Messages Bibliothek; beide sind nur für eine eingeschränkte Zahl von High-End Computern verfügbar.

[36] beschreibt die Prinzipien und den aktuellen Stand der Implementierung von Titanium. Titanium ist eine einfach zu erlernende Sprache, die dem Benutzer die Strukturiertheit moderner, objektorientierter Sprachen bietet. Sie unterstützt die Entwicklung explizit paralleler Programme und bietet einen optimierenden Compiler.

Titanium ist speziell für SPMD (single program multiple data) konzipiert. Parallelität entsteht durch die asynchrone Ausführung desselben Programms auf n Prozessoren. Unabhängig vom benutzten Backend stellt Titanium globale Referenzen zu Verfügung, mit denen lokale wie entfernte Objekte in gleicher Weise angesprochen werden können[3]. Dieses Vorgehen gibt dem Benutzer die Kontrolle (und auch die Verantwortung) über das Layout der Daten. Um dem Compiler die Optimierung zu erleichtern, können lokale Daten zudem als lokal deklariert werden.

Wichtige Erweiterungen gegenüber Java, auf die in den folgenden Abschnitten noch eingegangen wird, sind:

- Parallelität (Abschnitt 3.2)
- konstante Klassen, die nicht von `Object` abgeleitet sind (Abschnitt 3.4.2)
- Speichermanagement (Abschnitt 3.4.3)

[2] Soweit die Untersuchungen in Abschnitt 7 zeigen, erreicht Titanium dieses Ziel noch nicht. Dies mag jedoch an dem frühen Entwicklungsstadium liegen.

[3] In einer Sprache ohne direkte Zeigermanipulationen (vgl. Abnschnitt 2.7) entspricht das so erreichte Verhalten auf Maschinen mit verteiltem Speicher dem auf Maschinen mit gemeinsamen Speicher.

- echt mehrdimensionale Arrays und Domänen (Abschnitt 3.4.4)
- Überladen von Operatoren (Abschnitt 3.4.5)

3.1. Verfügbarkeit

Zum Test stand eine Vorabversion zur Verfügung, die von der Titanium Web-Seite in Berkeley[4] bezogen werden kann. Diese beinhaltet den Titanium-Compiler (`tcbuild`) und die benötigte Laufzeit-Bibliothek (tlib). Zusätzliche Bibliotheken, die für die verschiedenen Backends benötigt werden, müssen auf dem Zielrechner bereits installiert sein. Ein `configure` Script sorgt dafür, daß `tcbuild` die passenden Backends unterstützt.

Die Tests wurden mit dem `pthread` Backend durchgeführt. Zu bemerken ist, daß die Testversion bereits sehr stabil läuft. Titanium unterstützt verschiedene SMP-Plattformen (u.a. Solaris und Linux) und Arbeitsplatznetze mit spezieller Verbindungshardware. Die Unterstützung für Einprozessorsysteme ist vorwiegend zum Entwickeln und Testen der Programme gedacht.

3.2. Erzeugung von Parallelität

Parallelilität entsteht in Titanium durch die gleichzeitige Ausführung mehrerer Prozesse auf verschiedenen Rechnern in einem Netzwerk oder verschiedenen Prozessoren in einem Mehrprozessor-System. Zudem ist es möglich, Prozesse (pseudo-) parallel auf einem einzelnen Prozessor auszuführen. Die Anzahl der Prozesse bleibt während der gesamten Ausführungszeit konstant (statisches Prozeßmodell).

Die Anzahl der zu startenden Prozesse wird über eine Umgebungsvariable `TI_NODES` beim Programmstart festgelegt. Über `Ti.thisProc` und `Ti.numProcs` kann jeder Prozeß seine eigene Prozeßnummer, sowie die Anzahl aller laufenden Prozesse ermitteln. Die Art der benutzten Parallelität hängt vom entsprechenden Backend ab; dieses läßt sich beim Übersetzen als Parameter angeben. Änderungen am Programmtext sind dafür nicht erforderlich.

3.2.1. Threads

Die Titanium-Prozesse unterscheiden sich in einigen Punkten von Java-Threads. Während die Thread-Erzeugung in Java dynamisch und objektbasiert verläuft, ist in Titanium eine statische Anzahl von Prozessen während der gesamten Laufzeit verfügbar. Eine Methode um Threads, wie in Java, dynamisch zu erzeugen gibt es nicht. [36] weist unter *incompatibilities* darauf hin, daß Java-Threads in Titanium nicht implementiert wurden, erwähnt aber auch, daß es Überlegungen gibt, sie in eine spätere Version mit aufzunehmen.

[4]Die Titanium Homepage ist unter `http://www.cs.berkeley.edu/projects/titanium/` zu finden.

Obwohl der maximale Grad an (echter) Parallelität, der erreicht werden kann von der Anzahl der statisch erzeugten Prozesse abhängt[5], wäre es manchmal wünschenswert, längere und kürzere Berechnungen überlagern zu können, selbst wenn sich daraus keine Beschleunigung ergibt (z.B. um Benutzer-Eingaben während einer langen Berechnung zu ermöglichen). Diese Funktion soll in Titanium später eventuell von Java-Threads übernommen werden. Das bestehende Prozeßmodell bleibt davon weitgehend unberührt, was bedeutet, daß sich die Verteilung der Berechnung auf mehrere Prozessoren (oder Rechner) weiterhin auf die statisch erzeugten Titanium-Prozesse stützt, jeder diese Prozesse aber die Möglichkeit hat, dynamisch (leichtgewichtige) Java Prozesse zu erzeugen. Die Implementierung von Java-Threads würde es zudem ermöglichen, threadbasierte Java-Bibliotheken für Titanium-Programme nutzbar zu machen. Ein Beispiel für eine solche Bibliothek ist das AWT (Abstract Windowing Toolkit), mit dessen Hilfe sich grafische Benutzeroberflächen erstellen lassen.

Die Titanium Java-Bibliothek tlib enthält die Klassen und Schnittstellen, die zur Thread-Verwaltung in Java benötigt werden. Die zugehörige Maschinencode-Implementierung fehlt jedoch. Trotzdem erzeugt weder der Compiler, noch der Titanium-Linker eine Fehlermeldung bei dem Versuch dynamisch eigene Java-Threads zu erzeugen. Erst zur Laufzeit verweigert das Programm mit der Medlung `unimplemented native method: cl_java_lang_Thread_mt_currentThread` die weitere Ausführung. Dieses Verhalten ist unabhängig vom benutzten Backend, läßt aber hoffen, daß die Umsetzung des Java-Threadmodells bereits in Arbeit ist.

3.3. Synchronisation und Kommunikation

Aus dem Java-Standard wurde die Kommunikation über gemeinsamen Speicher sowie die Synchronisation über Monitore (vgl. Abschnitt 2.6.2) übernommen[6]. Auch die Methoden `wait` und `notify` stehen in Titanium zur Verfügung.

Zusätzlich zu den Methoden des Java-Standards ermöglicht Titanium eine eins zu N Kommunikation über `broadcast`, sowie eine N zu N Kommunikation über `exchange`. Eine `barrier`-Methode zur globalen Synchronisation ist ebenfalls verfügbar.

3.3.1. Barrier Synchronisation

Titanium erlaubt eine globale Synchronisation über Barriers. Jeder Prozeß P_i wartet an einem Methodenaufruf[7]

```
Ti.barrier();
```

[5] Wenn für jeden Prozeß ein eigener Prozessor zur Verfügung steht, ist der Grad an Parallelität gleich der Anzahl der Prozesse.

[6] Während in Java jedes beliebige Objekt als Monitor benutzt werden kann, sind in Titanium nur Objekte, die nicht als `immutable` deklariert sind als Monitore erlaubt.

[7] `Ti` ist dabei eine Klasse die von Titanium zur Verfügung gestellt wird. Sie definiert unter anderem die Methode `barrier`.

solange, bis alle anderen Prozesse seiner Prozeßgruppe den *textuell identischen* Methodenaufruf (d.h. die selbe Stelle im Programmtext) erreicht haben. Dies könnte jedoch dazu führen, daß ein Programm wie Abbildung 3.1 in einer Verklemmung endet, da jeder Prozeß P_i, $i = 0 \ldots n$ genau i mal an der ersten Barrier wartet und $n - i$ mal an der zweiten.

```
for(int i=0; i<numIterations; i++) {
  if (i<myProcessId) {
    //.. do some calculations
    Ti.barrier();
    }
  } else {
    //.. do some other calculations
    Ti.barrier()
    }
  }
```

Abbildung 3.1.: Eine sichere Verklemmung

Strukturelle Korrektheit

Damit solche Fehler bereits vom Compiler entdeckt werden können, definiert [36] den Begriff *strukturelle Korrektheit*.

Definition 1 (Strukturelle Korrektheit) *Ein Programm ist* strukturell korrekt*, wenn für alle seine Unterausdrücke ϵ folgendes gilt: Sei V die Menge aller einwertigen Variablen (genaueres hierzu folgt in Abschnitt 3.4.1) in ϵ. Wenn alle Prozesse die Ausführung von ϵ mit identischen Werten für jede Variable in V beginnen, und alle Prozesse terminieren, dann vollführen alle Prozesse dieselbe Folge von globalen Synchronisationen und beenden die Ausführung mit identischen Werten für jede Variable in V.*

Genaueres hierzu findet sich zudem in [3]. Die Analyse der strukturellen Korrektheit stützt sich auf die *einwertigen* (single valued) Variablen aus Abschnitt 3.4.1. Damit zur Übersetzungszeit festgestellt werden kann, ob die strukturelle Korrektheit verletzt wird, überprüft der Compiler, ob Anweisungen mit globalen Auswirkungen (wie z.B. ein Broadcast oder eine Barriere) möglicherweise nicht von allen Prozessen ausgeführt werden (können). Dieser Fall kann auftreten, wenn die Bedingung einer Schleife oder eines `if`-Konstruktes nicht single valued ist.

3.4. Besonderheiten

3.4.1. Einwertige Variablen

Einwertige Variablen in Titanium sind replizierte Variablen (jeder Prozeß besitzt eine eigene Instanz dieser Variablen), die in allen Prozessen denselben Wert haben. Um dies zu gewährleisten, muß die Verwendung dieser Variablen bestimmten Kriterien genügen, so dürfen als `single` deklarierte Variablen z.B. nur als Index für Schleifen benutzt werden, die in allen Prozessen synchron aufgezählt werden. Andererseits dürfen Anweisungen mit globalen Effekten (dazu zählen, neben den bereits genannten, auch Zuweisungen an einwertige Variablen) nur in Umgebungen benutzt werden, deren Kontrollfluß (Iterationen, Verzweigungen) ausschließlich von einwertigen Variablen oder Konstanten beeinflußt wird.

Der Compiler überwacht die richtige Verwendung von Variablen, die als `single` deklariert sind (vgl. Abschnitt 3.3.1). Neben Variablen dürfen auch Klassen und Methoden als `single` deklariert sein[8].

Einwertigen Variablen kommen zum Beispiel als Kontrollvariablen in der gemeinsamen (äußeren) Zeitschleife eines synchronen Programms (Abb. 3.2) zum Einsatz. Die Barrier-Synchronisation erzwingt dabei die synchrone Ausführung des Schleifenrumpfes, während die einwertige Kontrollvariable die strukturelle Korrektheit garantiert.

```
int single NUM_ITERATIONS=64;
for (int single i=0; i<NUM_ITERATIONS; i++) {
   // some calculation...
   Ti.Barrier();
}
```

Abbildung 3.2.: Ein synchrones Programm mit einwertigen Variablen

3.4.2. Unveränderbare Klassen

Wie in Abschnitt 2.4 bereits erwähnt, lassen sich alle skalaren Variablen in Java in geeignete Umschlagobjekte verpacken. Da durch die vorgegebene Klassenhierarchie die Umschlagobjekte aller numerischen Typen eine gemeinsame Basisklasse `java.lang.Number` besitzen, liese sich damit eine polymorphe Addiermethode wie folgt implementieren:

```
public Number myAdd(Number a, Number b) {
  return new Double(a.doubleValue() + b.doubleValue());
}
```

Das Ergebnis kann mit den Methoden aus `java.lang.Number` wieder in den gewünschten skalaren Typ konvertiert werden:

[8]Die Syntax, die von der aktuellen Version des Compilers verstanden wird: `int single a;` unterscheidet sich dabei etwas von der Syntax, die in der Titanium Dokumentation verwendet wird: `single int a;`

```
int a=2;
int b=3;
int sum = myAdd(new Integer(a), new Integer(b)).intValue();

// ergibt sum=5;
```

Dieses Beispiel macht bereits den Nachteil dieses Konzepts deutlich: Für eine Addition von zwei `int` Werten werden drei temporäre Objekte erzeugt. Wie für alle Objektparameter werden auch für die Umschlagklassen nur die Referenzen über den Laufzeitstack übergeben, obwohl diese Objekte konstant sind und ohnehin nicht verändert werden können. Da belegter Speicher in Java nicht explizit freigegeben werden kann, müssen solche Objekte während der Speicherbereinigung gefunden und aufgeräumt werden. Besser wäre es, die Objekte an Stelle der Referenzen auf dem Stack abzulegen. Die Kosten für die zusätzliche Indirektionsstufe der Wrapper-Klassen, die Speicherbereinigung und die Aufrufe der Destruktoren könnten so vermieden werden.

Titanium erlaubt genau dies durch die Verwendung von unveränderbaren Klassen (engl. *Immutable classes*). Klassen, die als `immutable` deklariert werden, sind von keiner anderen Klasse abgeleitet, und können selbst nicht weiter abgeleitet werden. Alle Variablen solcher Objekte sind entweder `static` (können also nach ihrer Initialiserung nicht mehr verändert werden) oder `final` (können in abgeleiteten Klassen nicht überladen werden). Dadurch können solche Objekte vom Compiler per Wert übergeben und direkt auf dem Stack abgelegt werden. Diese Vorgehensweise ist ein kleiner Bruch mit dem Objektmodell des Java-Standards, der ganz klar eine gemeinsame Basisklasse für alle Objekte vorsieht. Ein Zeiger auf `Object` ist in Java ein legaler "Universalzeiger", was wegen der strengen Typisierung sonst nicht möglich wäre. Auf der anderen Seite ist die Integration der skalaren Typen in Java ohnehin über einige Ecken gelöst, weshalb dem Leistungsgewinn an dieser Stelle durchaus Vorrang gewährt werden kann.

3.4.3. Speicherverwaltung und Referenzen

Titanium unterstützt das Konzept des *distributed shared memory*. Referenzen auf Objekte können grundsätzlich lokal oder global sein. Standardmäßig sind alle Referenzen global. Lokale Referenzen können mit entsprechenden Modifikatoren gekennzeichnet werden, um dem Compiler eine Optimierung zu ermöglichen.

Mit jedem Titanium-Prozeß ist genau eine *Region* fest verbunden, die dem diesem Prozeß zugeteilten Speicher entspricht. Neue Objekte befinden sich immer in der Region, die mit dem Prozeß verbunden ist, der sie erzeugt hat. Referenzen auf Objekte in anderen Regionen können über Kommunikationsprimitive bezogen werden, wie Abbildung 3.3 zeigt.

3.4.4. Arrays

Arrays sind in Java, wie in den meisten anderen Programmiersprachen, injektive Abbildungen von positiven ganzen Zahlen auf die entsprechenden Array-Elemente. Titanium ergänzt diese Funktionalität um sogenannte Gitter-Arrays (enlg. *Grid-Arrays*) oder kurz

```
class Example {
   Object referenced;
   Object reference;

   public static void main(String argv[]) {
      int myProc = Ti.thisProc();
      if (myProc == 0) {
         referenced = new AnyClass(); // create referenced instance
      }
      reference = broadcast referenced from 0; // 0 is the proc that owns
                                                // the instance
      reference.doSomething();
   }
}
```

Abbildung 3.3.: Bezug von Referenzen über Broadcasts in Titanium

Gitter (engl. *Grids*). Ein Gitter ist ein echt mehrdimensionales Array. Ein bestimmter Index eines Gitters wird über einen `Point` angegeben. Ein Punkt ist ein Tupel von Integern und läßt sich in Titanium wie folgt darstellen:

```
Point<3> p = [1, 1, 1];
Point<3> q = [42, 43, 112];
```

Der Wert in spitzen Klammern ist die Anzahl der Dimensionen dieses Punktes. Die Menge aller Punkte, also den Indexraum bestimmen dabei sogenannte Domänen (engl. *Domains*). Eine Domäne besteht grundsätzlich aus einer beliebigen Menge von Punkten, als Indexmengen für Arrays kommen in Titanium jedoch ausschließlich rechteckige Domänen zum Einsatz. Eine solche `RectDomain` wird einfach über ihre Eckpunkte definiert und enthält alle Punkte, die von dem aufgespannten Rechteck umschrieben werden:

```
RectDomain<3> r = [p : q];
```

Ein Array kann grundsätzlich über jeder rechteckigen Domäne definiert werden:

```
long [3d] a = new long[r];
```

`r` ist dabei die Domäne aus dem vorhergehenden Beispiel. Das Literal `3d` in eckigen Klammern hinter dem Typ-Bezeichner gibt die Anzahl der Dimensionen an. Diese Schreibweise ist zufällig identisch mit der Darstellung vom Fließkomma-Konstanten in Java, die jedoch sehr selten benutzt wird.

```
double value = 42.0d;
```

Da nur ganzzahlige Dimensionen zulässig sind, kann diese Mehrdeutigkeit leicht vom Compiler überprüft werden. Solange die Dimensionalität der Index-Parameter mit der Dimensionalität der Gitter-Deklaration übereinstimmt, lassen sich Gitter wie gewöhnliche Arrays verschachteln.

```
double[1d][2d] A = new double[D1][D2];
double v = A[p1][p2];
```

Dabei ist `D1` vom Typ `RectDomain<1>`, `D2` vom Typ `RectDomain<2>`, `p1` ist ein `Point<1>` und `p2` ein `Point<2>`.

Um dem Programmierer das Aufzählen von Gitter-Elementen zu erleichtern, wurde ein Iterator `foreach` eingeführt, der über alle Elemente der zugrundeliegenden Domäne iteriert. Ein kleines Beispielprogramm verdeutlicht die Verwendung des `foreach`-Befehls.

```
Point<3> p = [1, 1, 1];
Point<3> q = [3, 3, 3];
RectDomain<3> d = [p : q];

foreach(p in d) {
    System.out.println(p.toString());
}
```

Dieses Programm liefert folgende (gekürzte) Ausgabe. Die Reihenfolge in der die Punkte der entsprechenden Domäne aufgezählt werden, wirkt hier geordnet, ist aber in der Regel nicht vorhersagbar.

```
[1,1,1]
[1,1,2]
[1,1,3]
[1,2,1]
⋮
[3,2,3]
[3,3,1]
[3,3,2]
[3,3,3]
```

Zusätzlich zu den "normalen" Arrays, die Skalare oder Zeiger mit beliebigen Werten enthalten können, stellt Titanium mit sogenannten überlappenden Gittern (engl. *overlapping Grids*) Datenstrukturen zur Verfügung, die sich echt überlappen können, deren Elemente (ohne den Umweg über Zeiger) teilweise identisch sein dürfen. Textuell unterscheiden sich diese nicht von den bisher vorgestellten Titanium-Arrays. Dies führt natürlich dazu, daß der Compiler alle Arrays als potentiell überlappend behandeln muß, was einige Optimierungen erschwert. Um diese zu vereinfachen, gibt es eine Reihe zusätzlicher Regeln (und Operatoren) für überlappende Arrays:

- Formale Array Parameter für Methoden dürfen sich nicht überlappen, außer sie werden explizit als "überlappend" gekennzeichnet.

- Der Modifikator `overlap(F1,F2)` direkt hinter dem Kopf einer Methode kennzeichnet die formalen Parameter `F1` und `F2` dieser Methode als überlappend.

- Um an einer bestimmten Stelle im Programm auszudrücken, daß sich zwei Arrays `X` und `Y` nicht überlappen, kann der Aufruf `X.noOverlap(Y)` benutzt werden.

3.4.5. Überladen von Operatoren

Während in Java auf das Überladen von Operatoren verzichtet wurde[9], erlaubt Titanium das Überladen der folgenden Operatoren:

<table>
<tr><td>unäres Prefix</td><td>- ! ~</td></tr>
<tr><td>binär</td><td>< > <= >= == != + - * / & | ^ % << >> >>></td></tr>
<tr><td>Index</td><td>[] []=</td></tr>
<tr><td>Zuweisung</td><td>+= -= *= /= &= |= ^= %= <<= >>= >>>=</td></tr>
</table>

Das Überladen des Zuweisungsoperators `=` ist nicht möglich. Wenn ein Operator $\oplus$ überladen werden soll, genügt es, für die entsprechende Klasse `A` eine Methode `A.op`$\oplus$ mit zu definieren. Ähnlich wie in C++ gibt es keine Einschränkungen hinsichtlich der erlaubten Argument- oder Ergebnistypen. Ein Ausdruck `a` $\oplus$ `b` wird abgebildet auf `a.op`$\oplus$`(b)`; analoges gilt für unäre und Zuweisungsoperatoren. Ein Ausdruck `a[`e_1`,...,`e_n`]=b` wird zu `a.op[](`e_1`,...,`e_n`,b)`, der Ausdruck `a[`e_1`,...,`e_n`]` wird zu `a.op[](`e_1`,...,`e_n`)`.

Ein Konflikt tritt auf, wenn das rechte Argument von `+` vom Typ `String` ist[10], und für das linke Argument eine Methode `op+()` definiert wurde. In diesem Fall gibt Titanium der benutzerdefinierten Version den Vorzug.

[9]Eine Ausnahme bildet der Operator `+` für Strings, der vom Compiler gesondert behandelt wird. Eine Möglichkeit, dieses Verhalten zu beeinflussen, gibt es für den Programmierer nicht.

[10]Wie erwähnt, ist dies der einzige überladene Operator in Java.

4. HighPerformanceJava

Unter dem Namen *High Performance Java* gibt es wenigstens zwei verschiedene Projekte:

1. Die *Extreme Computing Group* am *Computer Science Department* der Indiana University will mit ihrem HighPerformanceJava Projekt die Eignung von Java für das belegen. Mit `javar` und `javab` standen zwei parallelisierende Java-nach-Java Compiler (einer für Quelltext, der andere für Bytecode) zum Test zur Verfügung.

2. Am *NPAC* (Northeast Parallel Architectures Center) der Syracuse University wird unter dem gleichen Namen unter anderem *HPJava*, ein Java-nach-Java Compiler entwickelt, der mittels eigener Bibliotheken den Sprachumfang von Java für SPMD-Architekturen erweitern soll.

Zur besseren Unterscheidung wird in dieser Arbeit das erste Projekt mit HighPerformanceJava, das zweite mit HPJava bezeichnet. Dieses Kapitel beschäftigt sich mit dem Projekt der Indiana University, HPJava ist Gegenstand von Kapitel 5.

Die Extreme Computing Group der Indiana University entwickelt unter dem Namen HighPerformanceJava unter anderem zwei restrukturierende Compiler für Java. Die Zielsetzung bei beiden Tools ist die Optimierung von Java-Programmen für Multiprozessor-Computer. Dabei wird implizite Parallelität im Quellprogramm erkannt[1] und in mehrere parallele Java-Threads umgesetzt. Die transformierten Programme entsprechen dem Java-Standard und sind ohne Einschränkung auf jeder javafähigen Plattform ausführbar. Eine Beschleunigung läßt sich mit dieser Methode jedoch nur auf Systemen erreichen, die Java-Threads auch tatsächlich parallel ausführen können (siehe auch Abschnitt 3.2). Beispiele für solche Architekturen sind das AIX4.2 JDK auf der IBM RS/6000 G3 [11], aber auch die SUN JDK 1.1 oder 1.2 Implementierung mit native Threads auf einer SPARC Multiprozessor Maschine, ebenso wie die Blackdown-Portierung des SUN JDK 1.1.7 oder 1.2 für Linux 2.2 auf Intel Multiprozessor-Systemen.

4.1. Verfügbarkeit

Zum Test standen Prototypen sowohl des Java-nach-Java Übersetzers `javar`, als auch des Bytecode Parallelisierers `javab` zur Verfügung. Beide Tools setzen auf Java als Zielplattform auf. Die von `javar` akzeptierte Syntax entspricht weitgehend dem Java-Standard Version 1.1.

[1] Beide Compiler werden als Prototyp eingestuft. Deshalb muß für javar implizite Parallelität derzeit explizit mit Kommentaren gekennzeichnet werden.

4.2. Erzeugung von Parallelität

Parallelität entsteht bei HighPerformanceJava durch die gleichzeitige Ausführung mehrerer Java-Threads. Im Moment kommen dafür hauptsächlich die bereits erwähnten SMP-Architekturen mit gemeinsam genutztem Speicher in Frage. Eine Möglichkeit diese Programme auf verteilten Systemen auszuführen, gibt es derzeit (mangels verteilter virtueller Maschinen) leider nicht[2].

Parallelisierbar sind unabhängige Schleifeniterationen und mehrfach rekursive Methodenaufrufe. Die Erkennung erfolgt entweder automatisch (`javab`) oder unterstützt durch den Programmierer (`javar`). Der Compiler überprüft in beiden Fällen, ob die ausgewählten Blöcke parallelisierbar sind. Eine Möglichkeit, Parallelität zu erzwingen, gibt es nicht[3]. Verschachtelte parallele Schleifen sind nicht erlaubt. Eine genauere Analyse der Parallelisierungsmöglichkeiten folgt in Abschnitt 4.4.

4.2.1. Details der Implementierung

Parallelisierbare Abschnitte werden in Form eines Iterationspools realisiert, der an die Idee des *Task-Farming* erinnert. In [8] wird die Implementierung genauer erläutert: Eine konstante Anzahl Threads werden als sogenannte *LoopWorker* (vgl. Abbildung 4.1) in einem Pool verwaltet. Jeder LoopWorker entnimmt eine Teilmenge der Iterationen der ursprünglichen (sequenziellen) Schleife aus dem zugeteilten `LoopPool`. Die Verteilung der Iterationen an die einzelnen Threads wird von der gewählten Scheduling-Methode bestimmt. Die Instanzvariablen `low`, `high`, und `stride` entsprechen der Ober- und Untergrenze sowie Schrittweite der aktuell zugeteilten Iterationen.

Vor Beginn der Ausführung einer parallelen Schleife wird in der aktuellen Version der Pool mit den LoopWorkern neu angelegt und initialisiert[4]. Die einzelnen Iterationen der Schleife werden ebenfalls in einem Pool verwaltet. Jeder LoopWorker entnimmt, abhängig von der gewählten Scheduling-Variante, eine oder mehrere Iterationen aus dem Schleifenpool, berechnet diese und terminiert, wenn sich keine unberechneten Iterationen mehr im Pool befinden.

Diese Vorgehensweise erlaubt zusammen mit dem passendem Scheduling eine nahezu optimale Lastverteilung und eignet sich gut für den allgemeinen, dynamischen Fall. Für den regulären Fall entsteht durch diese Methode ein deutlicher Overhead. Es gibt keine Möglichkeit das Layout der Daten oder den Ort der Ausführung zu beeinflussen, was auf den unterstützten Zielplattformen allerdings auch nicht nötig ist. SMP-Plattformen erlauben üblicherweise wahlfreie Kommunikation zu geringen Kosten. Die Allokation und die Zuteilung von Prozessoren erfolgt auf solchen Architekturen durch das Betriebssystem und ist im allgemeinen vom Programmierer nicht beeinflußbar.

[2] Zumindest ist dem Autor keine solche Implementierung bekannt.

[3] Da sich HighPerformanceJava auf den Java-Standard stützt, ist es natürlich trotzdem möglich neue Threads zu erzeugen. Da es sich dabei jedoch nicht um eine besondere Eigenschaft der untersuchten Sprache handelt, wird diese Möglichkeit hier nicht weiter untersucht.

[4] Diese Art der Implementierung ist nicht unbedingt optimal, wie in Abschnitt 4.4.1 näher erläutert wird.

```
abstract class LoopWorker extends Thread {
  int           low, high, stride;
  LoopPool      pool;
  RandomSync[]  sync;
  ...
}
```

Abbildung 4.1.: Die Klasse `LoopWorker` aus [8]

In [30] finden sich Untersuchungen mit ganz ähnlichen Implementierungen, die sich zwar nicht auf ein bestimmtes Parallelisierungswerkzeug beziehen, auf der Seite der erzeugten Programme jedoch durchaus vergleichbar sind. Diese bescheinigen der angewandten Parallelisierungstechnik eine gute Skalierbarkeit auf SMP-Architekturen und einen geringen Overhead auf Einprozessor-Maschinen. Es sei jedoch auch erwähnt, daß der Verwaltungsaufwand zur Erzeugung der Threads und des Schleifenpools nach in [30] aufgeführten Messungen in bestimmten Fällen eine Verschlechterung der Laufzeit um 20% bis 30% bewirken kann.

4.3. Synchronisation und Kommunikation

Die Kommunikation zwischen einzelnen Threads stützt sich ebenso wie die Thread-zu-Thread-Synchronisation auf die Mittel des Java-Standards. Diese wurden bereits im Detail in Abschnitt 2.6.2 vorgestellt. Basierend auf dieser Technik stellt HighPerformanceJava den einzelnen LoopWorkern einen gemeinsamen Vektor von Synchronisationsvariablen zur Verfügung. Jede dieser Synchronisationsvariablen enthält einen Bit-Vektor, dessen Länge der Anzahl der Schleifeniterationen entspricht. Mittels entsprechender `doPost`- und blockierender `doWait(i)`-Methodenaufrufe (`i` ist dabei der Schleifenindex) kann eine Synchronisation zwischen den Schleifeniterationen realisiert werden.

Anzumerken ist, daß die Implementierung der Methode `doWait(i)` im Prinzip dem Beispiel in Abbildung 2.1 entspricht.

4.4. Besonderheiten

4.4.1. `javar`

`javar` stellt die frühere der beiden Arbeiten dar. Das Programm arbeitet auf Quelltextebene und ist in der Lage, Schleifeniterationen und rekursive Methodenaufrufe auf mehrere Java-Threads zu verteilen (Abbildung 4.2 zeigt die Umwandlung von javar-Programmen in Bytecode). Die Erkennung von parallelisierbarem Code stützt sich dabei auf Anweisungen in Form von Kommentaren, wie das folgende Beispiel zeigt:

```
/*par threads = 16, cyclic */
for (int i=0; i<64; i++) {
```

```
    // ... some calculation
  }
```

Die Anzahl der Threads und die Art des Scheduling werden dabei explizit angegeben.

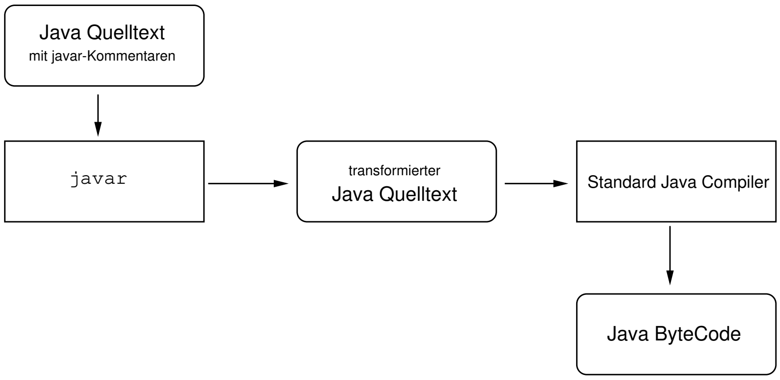

Abbildung 4.2.: Transformation von Programmen mit `javar`

Scheduling

`javar` unterstützt wahlweise zyklisches-, Block-, oder *kontrolliertes Selbstscheduling*(engl.*guided Selfscheduling*). Zyklisches Scheduling verteilt die Iterationen dabei "reihum", d.h. jeder Thread t führt die Iterationen $(i * T) + t$ aus, wobei T die Anzahl der Threads und i eine positive ganzzahlige Indexvariable ist. Beim Blockscheduling wird von jedem Thread t ein zusammenhängender Bereich I_t von Iterationen ausgeführt. Dabei ist $I_t = [t * (N/T), (t + 1) * (N/T)]$; T und N bezeichnen die Anzahl der Threads und die Problemgröße.

Beim kontrolliertem Selbstscheduling entnimmt jeder Thread Blöcke variabler Größe aus dem Pool. Die anfangs große Blockgröße wird im Verlauf der Abarbeitung sukzessive verkleinert. Das Verhalten entspricht in etwa dem eines statischen Blockschedulers, der sich allmählich in einen Selbstscheduler mit einer festen minimalen Blockgröße verwandelt.

Ein entsprechender Eintrag in dem einleitenden Kommentar vor dem zu parallelisierenden Block bestimmt, welche Art zum Einsatz kommt. Laut [30] erreicht das kontrollierte Selbstscheduling eine gute Lastverteilung mit relativ geringem Overhead. Diese Variante ist auch die Standard-Einstellung für das Scheduling in `javar`.

Mögliche Verbesserungen

Durch die objektorientierte Thread-Verwaltung in Java ist die Erzeugung neuer Threads eine teure Operation. Programme, die auf eine dynamische Thread-Erzeugung angewiesen sind, erfahren dadurch deutliche Leistungseinbußen. Die Verwaltung nicht benötigter

Threads in einem eigenen Pool ist eine gebräuchliche Methode, diesen Overhead zu reduzieren. Ein Thread-Pool findet sich auch in HighPerformanceJava, in der vorliegenden Implementierung wird der Pool jedoch vor jeder parallelen Schleife neu angelegt und anschließend nicht weiterverwendet. Abbildung 4.3 zeigt die von `javar` erzeugte Implementierung einer parallelen Schleife:

```
LoopWorker_Ludec_decomp_0[] worker_0 =
   (new LoopWorker_Ludec_decomp_0[128]);

for (int i_0 = 0; i_0 < 128; i_0++)
    worker_0[i_0] = (new LoopWorker_Ludec_decomp_0(a, l, u));

parlooppack.LoopWorker.parloop(1, n + 1, worker_0, 0, 2);
```

Abbildung 4.3.: `javar` Ausgabe für eine parallele Schleife

Der ursprüngliche Schleifenrumpf verbirgt sich hinter der Methode `parloop`. Wie deutlich zu sehen ist, wird vor Beginn der parallelen Schleife ein Array

```
LoopWorker_Ludec_decomp_0[]
```

von LoopWorkern erzeugt und in einer sequenziellen Schleife initialisiert.

Dies entspricht der Vorgabe, daß die Anzahl der zu verwendenden Threads im Kommentar vor dem zu parallelisierenden Block angegeben werden kann. Da dies auf SMP-Architekturen jedoch ohnehin eine Maschinenkonstante und im allgemeinen zumindest ein für größere Programmteile gleichbleibender Wert ist, erscheint diese Option unnötig. Sinnvoller wäre es, die Anzahl der Threads einmal für das ganze Programm (entweder an zentraler Stelle im Quelltext oder wie bei Message-Passing-Systemen beim Programmstart) festzulegen und den Thread-Pool während der gesamten Laufzeit beizubehalten.

Parallelisierung mehrfach rekursiver Methoden

Als einziges der betrachteten Projekte bietet HighPerformanceJava die Möglichkeit, neben Schleifeniterationen auch mehrfach rekursive Methoden zu parallelisieren. Dabei werden für mehrfache Aufrufe einer Methode mehrere Threads erzeugt.

```
void doSomething(int i) {

  doSomething(i+1);
  doSomething(i+1);

}
```

In diesem Beispiel würden somit die Aufrufe von `doSomething` auf zwei Threads verteilt. Eine Parallelisierung findet nur bis zu einer festgelegten Rekursionstiefe statt, darunter werden die Rekursionen sequenziell abgearbeitet. Eine Schleife, die als Endrekursion formuliert wird, läßt sich mit dieser Methode nicht parallelisieren, da auf jeder Rekursionsebene lediglich ein neuer Thread (für den Aufruf) erzeugt wird, während dessen Ausführung der erzeugende Thread in der Regel wartend gesetzt ist.

Da sich diese Arbeit auf die Parallelisierung von Schleifenprogrammen konzentriert, wird die Parallelisierung von Mehrfachrekursionen hier nicht genauer untersucht. Die Details hierzu finden sich in [8].

Anmerkungen und mögliche Erweiterungen

Die Annahme, daß nichtlokale Variablen durch *jeden* Methodenaufruf oder Zugriff auf Instanzvariablen verändert werden können, ist zwar korrekt, erscheint jedoch unnötig restriktiv. Wenigstens der Aufruf von Methoden aus den Standardbibliotheken wie `Math.ceil` oder einfacher Ein- Ausgabefunktionen wie `System.out.println` sollte dem Programmierer erlaubt sein.

Das Prinzip, nicht benötigte Java-Threads in einem zentralen Pool zur späteren Verwendung zwischenzuspeichern ist in der Java-Gemeinde weit verbreitet. Aus Effizienzgründen erscheint es sinnvoll, dieses Prinzip auch in HighPerformanceJava anzuwenden. In [17] finden sich Vorschläge, wie solche Pools für verteilte Systeme angepaßt werden können. Dadurch wäre es möglich, die Einschränkungen, die eine Festlegung auf SMP-Architekturen sicherlich mit sich bringt, zumindest teilweise zu umgehen.

4.4.2. `javab`

Unter dem Namen `javab` stellt die Indiana University den freien Prototyp eines Bytecode-nach-Bytecode Compilers zur Verfügung. javab ist selbst in C geschrieben und arbeitet auf der Ebene von bereits für die virtuelle Java-Maschine kompiliertem Bytecode. Ähnlich wie javar transformiert auch dieser Compiler implizit parallelen Quellcode in äquivalenten Zielcode, wobei Parallelität mit den im Java-Standard zur Verfügung stehenden Mitteln (mit der methode aus Abschnitt 4.2.1) erreicht wird. Der Unterschied zu javab besteht in der Fähigkeit, implizite Parallelität selbständig zu erkennen, ohne daß diese vom Programmierer gekennzeichnet wurde.

Da die Analyse auf Bytecode-Ebene stattfindet, bietet dieser Ansatz die interessante Perspektive, (Effizienz vorausgesetzt) in Just-In-Time Compiler oder andere Laufzeit-Optimierungssysteme integriert zu werden. Um die Laufzeit des Compilers zu verbessern, wurde auf eine vollständige Abhängigkeitsanalyse verzichtet; stattdessen werden kostengünstigere Abhängigkeitstests durchgeführt, um festzustellen ob Schleifeiterationen unabhängig sind.

Codeanalyse

Im Zuge der Codeanalyse wird zunächst eine Kontrollfluß- und anschließend eine Datenflußanalyse des Quellprogramms durchgeführt. Im Gegensatz zu den Analysemethoden für andere Zwischencode-Darstellungen (z.B. Drei-Adress-Code), die allgemein bekannt sind, kommt für JVM-Bytecode ein eigenes Verfahren [11] zum Einsatz, das den Besonderheiten der stackbasierten Maschinenbefehle Rechnung trägt. Das Ergebnis dieser ersten Analyse ist ein Flußgraph $G = \langle V, E, v_0 \rangle$ mit Startknoten v_0, der als Basis für die Erkennung unabhängiger Schleifendurchgänge dient.

Um die Klasse der mit dieser Methode parallelisierbaren Schleifen beschreiben zu können, müssen zunächst einige Begriffe eingeführt werden: Die parallelisierbaren Schleifen sind eine Teilmenge der *trivialen Schleifen* und diese wiederum sind eine Teilmenge der *natürlichen Schleifen*. Letztere werden mit Hilfe von *Dominatoren* und *Rückwärtskanten* beschrieben, wobei deren Definition in [11] weitgehend mit der in [2] übereinstimmt.

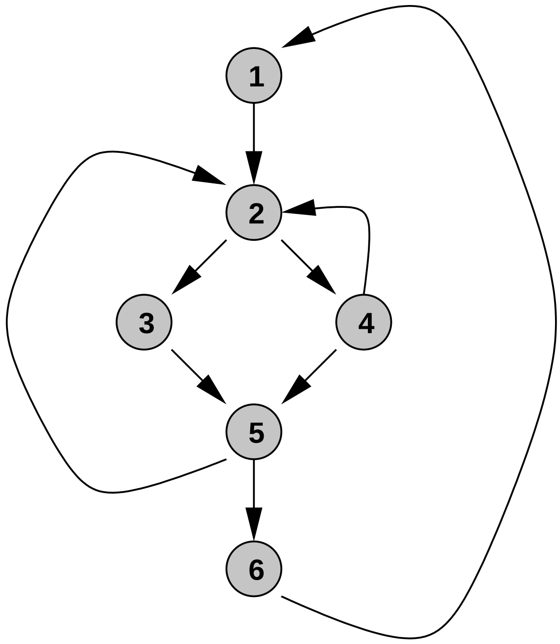

Abbildung 4.4.: Flußgraph

Definition 2 (Dominator) *Ein Knoten $w \in V$ eines Flußgraphen $G = \langle V, E, v_0 \rangle$ dominiert einen Knoten $v \in V$ wenn jeder gerichtete Pfad vom Startknoten v_0 zu v den Knoten w enthält (Der Startknoten dominiert somit alle Knoten des Graphen). Die Menge aller Dominatoren eines Knoten $w \in V$ wird mit $Dom(w)$ bezeichnet.*

Abbildung 4.4 zeigt einen Flußgraphen. In diesem Beispiel dominiert Knoten 1 (der Startknoten) alle anderen Knoten, Knoten 2 dominiert alle Knoten außer 1, Knoten 3, 4 und 6 dominieren nur sich selbst, und Knoten 5 dominiert Knoten 6.

Mit Hilfe dieser Dominatorinformation lassen sich die natürlichen Schleifen im Flußgraphen identifizieren.

Definition 3 (Natürliche Schleife) *Eine Schleife $L \subseteq V$ heißt natürliche Schleife, wenn gilt:*

- *Es gibt genau einen Einstiegspunkt in die Schleife. Dieser Knoten dominiert alle Knoten der Schleife.*
- *Es existiert mindestens ein Weg zum Iterieren der Schleife, es gibt also eine Kante, die von einem Knoten der Schleife zurück zum Einstiegspunkt führt.*

In dem betrachteten Beispiel bilden die Knoten 2, 3, 4 und 5 eine natürliche Schleife. Der Knoten 2 ist der Einstiegspunkt (oder *Kopf*) der Schleife, während die Kante von Knoten 5 zu Knoten 2 eine sogenannte Rückwärtskannte darstellt.

Definition 4 (Rückwärtskante) *Eine Rückwärtskante ist eine Kante, die von einem Knoten $g \in V$ zu einem Dominator $h \in V$ von g zurückführt, also eine Kante $(g, h) \in E$ mit $h \in Dom(g)$.*

Neben einem Einstiegspunkt besitzt eine natürliche Schleife einen[5] oder mehrere Ausgänge, die wie folgt definiert sind:

Definition 5 (Ausgang einer Schleife) *Eine Kante $(v, e) \in E$ einer natürlichen Schleife $L \subseteq V$ mit $v \in L$ und $e \notin L$ heißt* Ausgang *der Schleife. Ein solcher Ausgang ist entweder regulär oder irregulär, abhängig davon, ob die entsprechende Kante ein Übergang während der normalen Programmausführung ist, oder ob die Kante direkt auf eine Anweisung folgt, die eine Ausnahme verursacht.*

Über Anzahl und Art der Schleifenausgänge läßt sich nun eine besondere Teilmenge der natürlichen Schleifen definieren, die sogenannten trivialen Schleifen:

Definition 6 (Triviale Schleife) *Eine natürliche Schleife $L \in V$ in einem gegebenen Flußgraphen $G = \langle V, E, v_0 \rangle$ heißt* triviale Schleife *mit Schleifenindex i, wenn gilt [11]:*

i. Es gibt genau einen regulären Schleifenausgang $(h, e) \in E$. Dieser Ausgang ist Alternative eines Konditionals, welches i mit einem Ausdruck vergleicht, der von keiner Anweisung $v \in L$ verändert wird.

ii. Die letzte Anweisung $v \in L$, die keinen Sprungbefehl darstellt, ist von der Form $i := i + p$ wobei $p > 0$.

Diese Bedingung wird sowohl von `for`- als auch von `while`-Schleifen erfüllt. Da in Java ebenso wie in C die Semantik von `for` und `while` Schleifen sehr ähnlich ist, ist eine Unterscheidung auf Bytecodeebene ohnehin kaum möglich.

Eine triviale Schleife ist parallelisierbar, falls gilt:

i. Der Operandenstack ist am Ein- und Ausgang der Schleife leer.

[5] Natürlich sind auch Schleifen ohne Ausgang denkbar, allerdings sollte man beim Auftreten solcher Endlosschleifen seinen Programmierstil überdenken. Solche Schleifen werden durch eines der folgenden Kriterien von der Parallelisierung ausgenommen.

ii. Der Rumpf der Schleife enthält keine Methodenaufrufe (also auch keine Objektinstanziierungen) und keine schreibenden Zugriffe auf Instanzvariablen oder `static` Variablen.

iii. Der Compiler kann sicherstellen, daß die Schleife nicht irregulär beendet wird (siehe unten).

iv. Alle Schleifendurchgänge sind unabhängig (siehe unten).

Die Bedingungen i und ii können vom Compiler leicht überprüft werden:

- Bedingung i folgt direkt aus der durchgeführten Berechnung der Stackzustände [11].
- Bedingung ii ist erfüllt, wenn der Bytecode für den Schleifenrumpf keinen der Befehle `invoke<kind>`, `putfield` oder `putstatic` enthält.

Die Analyse der Bedingungen iii und iv sind Inhalt der folgenden Abschnitte.

Behandlung von Ausnahmen Die Semantik von Java-Programmen hängt u.a. von einer korrekten Behandlung der Ausnahmen ab. Da diese Behandlung jedoch eng an den Laufzeitstack des aktuellen Threads gebunden ist, ist es schwer, die Semantik in einer parallelisierten Version beizubehalten. javab trägt diesem Umstand Rechnung, indem nur Blöcke für die Parallelisierung in Betracht kommen, für die das Auftreten von Ausnahmen bereits zur Übersetzungszeit ausgeschlossen werden kann. Über einen Aufrufparameter läßt sich diese Restriktion ein wenig aufweichen, indem auf den Erhalt der Semantik bei Linkerfehlern verzichtet wird.

Für das Auftreten von Laufzeit- oder Linkerfehlern gelten folgende Regeln:

- Eine Integer-Division oder Modulus-Berechnung verursacht keinen Fehler, wenn der zweite Operand (also das erste Element auf dem Operandenstack) ungleich 0 ist.
- Der Zugriff auf eine Instanzvariable verursacht keinen Fehler, wenn der Zugriff aus derselben Instanz erfolgt, zu der auch die Variable gehört.
- Ein Arrayzugriff verursacht keine Ausnahme, wenn der Zeiger auf das Array ungleich `null` ist, und sich der Index innerhalb der Arraygrenzen befindet.

Der letzte Punkt ist besonders interessant, da sich diese Bedingung nur für bestimmte Fälle ohne Laufzeittests (vgl. Abschnitt 2.10) garantieren läßt.

Erkennung unabhängiger Schleifendurchgänge Zur Erkennung unabhängiger Schleifendurchgänge ist es zunächst notwendig, zwischen der *Definition* und *Verwendung* einer Variable zu Unterscheiden. Eine Anweisung, die den Wert einer Variablen i benutzt, wird als Verwendung, eine Anweisung, die den Wert von i verändert, wird als Definition bezeichnet. Die Zuweisung

```
a = b
```

stellt somit eine Verwendung von `b` und eine Definition von `a` dar. Man sagt, eine Definition d einer Variable i *erreicht* eine Verwendung u von i, wenn es im Flußgraphen einen Pfad von d nach u gibt, auf dem i nicht verändert wird. In [2, Kap. 10] werden in diesem Zusammenhang zwei Mengen beschrieben, die sogenannten Verwendungs-Definitions-Ketten (UD-Ketten) und Definitions-Verwendungs-Ketten (DU-Ketten).

Die Menge $\mathrm{UD}(u)$ enthält alle Definitionen d von i, die die Verwendung u erreichen. Umgekehrt enthält die Menge $\mathrm{DU}(d)$ alle Verwendungen u von i, die von der Definition d erreicht werden.

Die Durchgänge einer Schleife sind unabhängig, wenn ihre Variablen keine Abhängigkeiten erzeugen. Diese Anforderung läßt sich mit Hilfe der vorangegangenen Definition wie folgt formulieren: Die lokalen skalaren Variablen einer Schleife $L \subseteq V$ erzeugen keine Abhängigkeiten wenn gilt [11]:

1. Für jeden lesenden Zugriff $u \in L$ auf eine lokale Variable i (außer dem Schleifenindex) gilt entweder:

   a) Für alle Definitionen $d' \in \mathrm{UD}(u)$ ist $d' \notin L$, oder

   b) es gibt eine Definition $d' \in \mathrm{UD}(u)$ und $d' \in L$, sodaß $d' \in Dom(u)$ und $d' \neq u$.

2. Für jede Anweisung $d \in L$, die die Definition einer lokalen Variable j (einschließlich des Schleifenindex) darstellt, gibt es keine Anweisung $v' \notin L$ die einer Verwendung $u' \in \mathrm{DU}(d)$ dieser Variable entspricht.

Diese Bedingung bezieht sich (wie gesagt) auf skalare Variablen; für die Parallelisierung von trivialen Schleifen, die Zugriffe auf Arrays enthalten, müssen zudem zwei weitere Bedingungen erfüllt sein:

1. Die Arrayindizes müssen bei jedem Zugriff innerhalb der Indexgrenzen liegen, da sonst die Schleife durch eine Ausnahme anormal beendet wird.

2. Durch die Zugriffe auf Arrayelemente dürfen keine schleifengetragenen Abhängigkeiten entstehen.

Um die erste dieser Bedingungen zu überprüfen, berechnet der Compiler die Unter- und Obergrenze von Indexausdrücken in Abhängigkeit von lokalen Variablen. Betrachtet werden hierbei Ausdrücke der Form $c+i$, wobei c eine Konstante und i eine lokale Variable ist. Wenn sich die Größe des Arrays nicht feststellen läßt (z.B. weil es nicht im aktuellen Block instantiiert wurde), wird eine entsprechende Frage an den Benutzer ausgegeben oder die Schleife von der Parallelisierung ausgenommen[6]. Abbildung 4.5 zeigt die Fragen an den Benutzer während der Analyse der folgenden Schleife:

```
static void query(int[][] a, int[] b, int x[], int n) {
   for (int i=0; i<=n+100; i++) {
      for (int j=0; j<n-20; j++) {
```

[6]Ein Aufrufparameter legt fest, ob Benutzerinteraktion gewünscht ist, oder nicht

```
                b[i] += a[i][j] * x[j];
            }
        }
    }
```

```
*** Loop.class:: Queries for method 'query([[I[I[II)V'

  > Queries for the 2-dimensional array parameter '[[I' in local 0
    * dim-1: is range [0,101+n) valid where n = parameter in local 3?
    * dim-2: is range [0,-20+n) valid where n = parameter in local 3?

  > Queries for the 1-dimensional array parameter '[I' in local 2
    * dim-1: is range [0,-20+n) valid where n = parameter in local 3?

  > Queries for the 1-dimensional array parameter '[I' in local 1
    * dim-1: is range [0,101+n) valid where n = parameter in local 3?

Is storage of array parameters 0 '[[I' and 1 '[I' non-overlapping?

Is storage of array parameters 2 '[I' and 1 '[I' non-overlapping?
```

Abbildung 4.5.: Benutzerinteraktion mit `javab`

Der Compiler erzeugt für jeden m-dimensionalen Array-Parameter einen Zugriffsvektor $\overline{x} = \langle x_0, \ldots, x_{m-1} \rangle$. Die Elemente x_j des Vektors geben Auskunft über die Zugriffe auf die entsprechende Dimension des Arrays. Sei i die Indexvariable der untersuchten Schleife, so gilt:

$$x_j = \begin{cases} \top & \text{Dimension } j+1 \text{ wird nicht referenziert} \\ [l,u] & \text{Dimension } j+1 \text{ wird durch } i+c,\ c \in [l,u] \text{ referenziert} \\ \perp & \text{Dimension } j+1 \text{ wird durch beliebigen Ausdruck referenziert} \end{cases} \tag{4.1}$$

Für jede Zuweisung an ein Arrayelement innerhalb der Schleife überprüft der Compiler, ob für wenigstens ein Element x_j des Zugriffsvektors $\overline{x}$ gilt: $x_j = [l,u]$ und $u - l < p$. Dabei ist p die Schrittweite der Schleife. Gibt es kein solches Element, so kann die Existenz schleifengetragener Abhängigkeiten nicht ausgeschlossen werden und die Schleife wird nicht parallelisiert.

Implementierung der Parallelität

Schleifen, die als parallelisierbar identifiziert wurden, werden nach der Methode aus Abschnitt 4.2.1 parallelisiert. Wie eigene Untersuchungen des generierten Bytecodes zeigen, werden für jede parallelisierbare Schleife vier LoopWorker Threads erzeugt, die vollständig asynchron laufen. Die Zuteilung der Iterationen erfolgt dabei zyklisch, allerdings ohne den Umweg über einen zentralen Thread-Pool.

5. HPJava

HPJava [38] ist eine Java-Erweiterung, die am NPAC der Syracuse University entwickelt wird. Ziel dieser Arbeit, die offensichtlich stark von früheren Arbeiten zu High Performance Fortran (HPF[26]) beeinflußt wurde, ist die Bereitstellung einer Arbeitsumgebung für wissenschaftliche Parallelprogrammierung auf der Grundlage von Java. Zu diesem Zweck wird ein Precompiler und eine Reihe von Laufzeitbibliotheken vorgestellt [14], die die Erstellung datenparalleler Programme ermöglichen sollen.

Single Program Multiple Data (SPMD) steht bei HPJava vorwiegend für die Programmierung mit verteilten Arrays und ist der angestrebte Programmierstil, den die Erweiterungen unterstützen sollen. Die Verteilung soll dabei transparent erfolgen; neue Sprachprimitive erlauben eine Steuerung durch den Programmierer auf relativ abstrakter Ebene. Komplexe Kommunikationsmuster stehen in Form (deadlockfreier [38]) Laufzeitbibliotheken zur Verfügung; gleichzeitig sind auch direkte Aufrufe der zugrundeliegenden Kommunikationsbibliotheken möglich. Einige parallele Kontrollkonstrukte sowie globale Variablen runden das Paket ab.

Obwohl sich HPJava zum Teil noch in der Spezifikationsphase befindet, erscheinen die (geplanten) Leistungsmerkmale vielversprechend genug, um das Projekt in diese Arbeit mit aufzunehmen. Da keine Version zum Test zur Verfügung steht, beschränkt sich dieses Kapitel auf Aussagen der entsprechenden Publikationen.

Die wichtigsten Erweiterungen gegenüber dem Java-Standard sind:

- Prozeß-Gruppen und -Arrays (Abschnitt 5.4.1)
- Globale Variablen und verteilte Arrays (Abschnitt 5.4.2 und 5.4.3)
- drei parallele Sprachkonstrukte, darunter eine datenparallele Schleife (Abschnitt 5.4.4)

Ein Java-nach-Java Compiler (im weiteren oft Frontend) genannt, ersetzt die wenigen Syntaxerweiterungen durch geeignete Bibliotheksaufrufe. Zudem werden beim Übersetzen einfache Semantikprüfungen durchgeführt und der Code um geeignete Laufzeittests ergänzt, um z.B. die Einhaltung von Array-Grenzen zu gewährleisten. Die Zielsprache des Precompilers ist Java, Ausführungsplattform ist somit jede beliebige Java-VM,[1] vorausgesetzt, daß die zusätzlich benötigten Bibliotheken für das zugrundeliegende System verfügbar sind.

[1] Im Unterschied zu HighPerformanceJava sind auch mehrere kooperierende VMs auf verschiedenen Rechnern als Zielplattform möglich.

Teile der Bibliotheken sind in C/C++ realisiert und bieten somit Zugriff auf bestehende Kommunikationsbibliotheken in Maschinencode. Zu diesen zählen Standard-Kommunikationsbibliotheken wie MPI und eine (ursprünglich für HPF entwickelte) universelle Kommunikationsbibliothek .

5.1. Verfügbarkeit

HPJava befindet sich noch in der Entwicklung weshalb viele Teile (insbesondere das Frontend) noch nicht verfügbar sind. Es war jedoch möglich, eine funktionsfähige Version von MPIJava, der Java-Schnittstelle zu MPI zu bekommen. Die Kommunikationsbibliothek Adlib ist ebenfalls bereits verfügbar, allerdings ist die benötigte Java-Schnittstelle AdJava noch nicht freigegeben. Der Grund dafür ist u.a., daß AdJava eng mit dem noch nicht fertiggestellten Frontend zusammenarbeitet und dadurch wohl noch größeren Änderungen unterworfen sein wird.

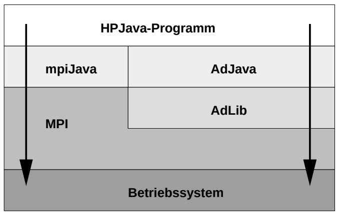

Abbildung 5.1.: Schichtenmodell für MPI-Aufrufe durch HPJava

Die Bibliotheken werden als Source-Code vertrieben und lassen sich auf den meisten Systemen übersetzen, für die die zugrundeliegenden Kommunikationsbibliotheken verfügbar sind. Die Webseite des *PCRC* nennt u.a.: Solaris, IRIX und NT. Aus eigener Erfahrung ist auch die Übersetzung unter Linux kein Problem. Im Augenblick wird an MPIJava am intensivsten gearbeitet; zu diesem Thema sind auch die aktuellsten Publikationen erschienen. Wie auf Anfrage zu erfahren war, wird auch das Frontend kontinuierlich weiterentwickelt und soll in jedem Fall fertiggestellt werden.

5.2. Erzeugung von Parallelität

Ebenso wie bei HighPerformance Java (vgl. Abschnitt 4.2) entsteht Parallelität durch die gleichzeitige Ausführung mehrerer Java-Threads. Durch die Verwendung von Standard-Kommunikationsbibliotheken ist es hier allerdings möglich, Threads auf verschiedenen (also auch verteilten) Rechnern auszuführen. Die Einschränkung, daß sich alle Threads innerhalb derselben virtuellen Java-Maschine befinden müssen, entfällt, wodurch sich die Menge der verfügbaren Zielarchitekturen erheblich vergrößert. Voraussetzung ist, daß sowohl die Kommunikationsbibliotheken als auch eine Java-VM für jeden einzelnen Rechner verfügbar sein müssen.

Das Prozeßmodell ist (durch die Message-Passing-Implementierung) statisch, allerdings stehen die Methoden zur dynamischen Thread-Erzeugung dem Programmierer uneingeschränkt zur Verfügung. Während sich die Implementierung der verteilten Datenstrukturen auf das statische Modell stützt, können hier z.B. Benutzer-Ein- Ausgabe und Berechnung mittels dynamisch erzeugter Threads überlagert werden.

5.3. Synchronisation und Kommunikation

Durch die gemeinsame Zielplattform stehen HPJava, ebenso wie HighPerformanceJava (vgl. Abschnitt 4.3) die Methoden des Java-Standards zur Verfügung, welche in Abschnitt 2.6.2 bereits vorgestellt wurden.

Die Kommunikation erfolgt in der Regel über verteilte Datentypen (im wesentlichen verteilte Arrays vgl. Abschnitt 5.4.2). Da die Kommunikation über gemeinsame Datenobjekte starken Einschränkungen unterworfen ist (z.B. dürfen Zugriffe auf verteilte Arrays keine Kommunikation verursachen), gibt es zum Ausgleich eine leistungsfähige Kommunikationsbibliothek, die allgemeine Operationen wie `shift`, `sum`, `copy`, etc. realisiert.

Für eine Kommunikation mit den nächsten Nachbarn lassen sich verteilte Arrays zudem mit sogenannten *Halos* umgeben. Dadurch kann eine Umgebung beliebiger Größe um den lokalen Ausschnitt redundant gespeichert und über Bibliotheksaufrufe (`Adlib.writeHalo`) aktualisiert werden.

5.4. Besonderheiten

5.4.1. Prozesse und Prozeßgruppen

Wie bei SPMD üblich, wird ein HPJava-Programm gleichzeitig von mehreren Prozessen ausgeführt, wobei ein Prozeß im wesentlichen einer virtuellen Maschine entspricht. Diese Prozesse werden in Prozessgruppen verwaltet, die nicht notwendigerweise disjunkt sein müssen. Zwar gibt es für jeden Prozeß zu jedem Zeitpunkt genau eine wohldefinierte *aktive Prozessgruppe*, jedoch läßt sich diese über spezielle Anweisungen jederzeit ändern. Prozeßgruppen habe in der Regel die Struktur von Arrays und werden durch entsprechende Klassen repräsentiert.

`Group` ist die Basisklasse für alle Prozeßgruppen. Davon abgeleitet existieren verschiedene Klassen, die Prozeßarrays unterschiedlicher Dimensionalität repräsentieren. Zum Beispiel ist

```
Procs2 p = new Procs2(2,4);
```

ein zweidimensionales 2×4 Array von Prozessen. Ein vordefiniertes, eindimensionales Prozessarray `Procs1 Adlib.global` ist immer vorhanden. Diese Gruppe enthält alle Prozesse der aktuellen Laufzeitumgebung und ist die aktive Prozessgruppe nach dem Start des Programms. Der Befehl `on` kann benutzt werden, um die aktive Prozessgruppe während der Ausführung zu wechseln:

```
Procs2 p = new Procs2(2,4);
on(p) {
  ... some code
}
```

Prozessgruppen spielen auch eine Rolle bei der Speicherung von verteilten und/oder globalen Arrays (vgl. Abschnitt 5.4.2 und 5.4.3). Die Abbildung von Array-Elementen auf Prozesse einer `Group` geschieht dabei über sogenannte *Ranges*. Ein `Range` bildet ein Integer-Intervall auf eine Prozeßdimension ab. Dabei sind verschiedene Verteilungsmuster als abgeleitete Klassen bereits vordefiniert:

```
Range x = new BlockRange(100, p.dim(0));
Range y = new CyclicRange(200, p.dim(1));
```

5.4.2. Arrays

Neben den Java-Arrays gibt es in HPJava eine Klasse von mehrdimensionalen und verteilten Arrays als eigenständiges Sprachmittel. Diese unterscheiden sich schon in der Syntax von ihrem Java-Gegenstück. Zum Beispiel definiert

```
int[[,]] a = new int [[5,5]];
```

ein zweidimensionales Array von Integern. Die doppelten Klammern kennzeichnen die HPJava-Erweiterung. Der Zugriff auf ein einzelnes Element wird durch einfache Klammern gekennzeichnet:

```
int b = a[2,3];
```

im Gegensatz zu den Java-Arrays erlaubt HPJava sogenanntes *Regular-Section-Subscripting*. Diese Art von Indizierung wird durch doppelte Klammern ausgedrückt. Das Ergebnis ist kein einzelnes Element, sondern ein Array von passender Dimensionalität. So liefert

```
int [[]] c = a[[0,:]]
```

die erste Spalte von `a` als eindimensionales Array `c` zurück. Die Definition der Indexbereiche darf wie in Fortran90 eine (jeweils optionale) Ober- und Untergrenze, sowie (ebenfalls optional) eine Schrittweite enthalten. Die Schreibweise als durch Doppelpunkte getrennte Tripel ist wie in Fortran:

```
int [[]] d = a[[0,1:4:2]]
```

Wie bereits erwähnt, können HPJava-Arrays über Prozeßgruppen verteilt werden. Dazu wird bei der Definition die verteilte Dimension mit “`#`” markiert:

```
Procs2 p = new Procs2(3,2);
Range x = new BlockRange(100, p.dim(0));
Range y = new BlockRange(200, p.dim(1));

float [[#,#]] a = new float [[x,y]] on p;
float [[#,#,]] b = new float [[x,y,300]] on p;
```

Im Gegensatz zu den meisten anderen parallelen Sprachen impliziert eine Array-Indizierung in HPJava niemals Kommunikation zwischen zwei Prozessen. Um dem Programmierer den Umgang mit diesem Detail zu vereinfachen, wurden für den Zugriff auf Array-Elemente sogenannte *Locations* eingeführt. Ein Element eines `Range` wird als *Location* oder *Slot* bezeichnet und kann zum Zugriff auf das entsprechende Arrayelement benutzt werden.

```
Location i = x[17], j = y[23];
```

Die Locations `i` und `j` repräsentieren demnach die Prozesskoordinaten des Prozesses, für den das Element `a[17,23]` lokal ist. Nur dieser Prozeß darf direkt über Integer-Indizes auf das Element zugreifen. Um dem Programmierer diese in der Regel aufwendigen Fallunterscheidungen zu ersparen, wurde das Schlüsselwort `at` eingeführt:

```
on(p)
  at(i) {
    at(j) {
      a[17,23] = 5;   // lokaler Zugriff über Integer-Indizes
      // oder
      a[i,j] = 5;     // lokaler Zugriff über Locations
    }
  }
}
```

Der Block eines `at(i)` Ausdrucks wird nur von den Prozessen der Gruppe `p` ausgeführt, welche die Location `i` lokal speichern.

5.4.3. Globale Variablen

Diese (eben vorgestellte) "abschnittsweise lokale" Speicherung verteilter Arrays ist konsistent mit der Sichtbarkeit von Variablen in Java. Jede virtuelle Maschine (und dementsprechend jeder HPJava-Prozeß) definiert ihren eigenen Adress- und Namensraum. Alle Variablen in HPJava sind in diesem Sinne zunächst lokal. Zusätzlich gibt es jedoch die Möglichkeit, auch *globale Variablen* zu erzeugen; die Details hierzu finden sich in [38].

Prinzipiell wird für jede globale Variable ein *Deskriptor* erzeugt, der einen Verweis auf die Prozessgruppe und den Speicherort der Variablen enthält. Dieser Deskriptor wird von allen Prozessen anstelle der Variablen selbst benutzt, um auf ihren Wert zuzugreifen. Die Semantik entspricht somit etwa der der einwertigen Variablen von Titanium aus

Abschnitt 3.4.1. Die Syntax der Deklaration und Verwendung solcher Variablen ist etwas gewöhnungsbedürftig:

```
on(p) {
  int # s = new int #;

  # s = 100;
}
```

zumal “#” hier die Funktion eines Präfixoperators erfüllt, der den *Handle* `s` in den benötigten Deskriptor überführt. Da dieses Konzept in neueren Veröffentlichungen nicht mehr erwähnt wird, und die globalen Arrays aus [38] in wesentlichen Punkten den in Abschnitt 5.4.2 vorgestellten verteilten Arrays entsprechen, wird auf eine eingehendere Betrachtung an dieser Stelle verzichtet[2]. Der interessierte Leser sein stattdessen auf die entsprechende Veröffentlichung verwiesen.

5.4.4. Parallele Schleifen

Als logische Konsequenz aus dem `at`-Konstrukt, folgt eine datenparallele Schleife, die mit `over` eingeleitet wird:

```
Index i = new Index(x), j = new Index(y);
over(i) {
  over(j) {
    a[i,j] = 2 * a[i,j];
  }
}
```

Der Parameter für `over` ist ein `Index` und entspricht der Menge der verschiedenen `Locations` des zugrundeliegenden `Ranges`. Das obige Beispiel zeigt eine parallele Schleife, die jedes Element von `a` mit 2 multipliziert. Dabei führen alle Prozesse der Gruppe `p` die Berechnung für die jeweils lokalen Elemente von `a` parallel aus.

[2] Möglicherweise wird auch im Laufe der Entwicklung auf globale Variablen zugunsten verteilter Arrays und leistungsfähiger Kommunikationsbibliotheken verzichtet werden.

6. JavaParty

JavaParty wird am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelt. Der Name ist hierbei Konzept: Mehrere kooperierende virtuelle Java-Maschinen (verteilt auf mehrere Rechner) agieren aus Programmierersicht wie eine einzige VM mit gemeinsamen Speicher. Eine zentrale Instanz übernimmt die Verwaltung; zudem können sich jederzeit neue Rechner an der "Party" beteiligen, oder diese verlassen. Neu erzeugte Objekte werden entweder automatisch oder durch den Programmierer einzelnen Maschinen zugeordnet. Diese interagieren mit Objekten auf anderen Maschinen für den Benutzer transparent über die Java RMI-Schnittstelle (vgl. Abschnitt 2.9).

Das System ist vollständig in Java realisiert und somit auf jeder javafähigen Plattform einsetzbar. Die als Backend benutzte RMI-Schnittstelle ist in jeder Java-Distribution enthalten. Alternativ stehen für JavaParty auch geringfügig veränderte RMI-Basisklassen zur Verfügung, welche (anders als Standard-RMI) nicht davon abhängig sind, daß die Netzwerkinfrastruktur Sockets zur Verfügung stellt. So unterstützt JavaParty z.B. auch die ebenfalls in Karlsruhe entwickelte ParaStation Technologie[1], die es erlaubt, bis zu 4096 Arbeitsplatzrechner zu einem Großrechner zu kombinieren.

Wichtiges Entwurfsziel war die Ausnutzung von Lokalität. Ein Nachteil bei der Erstellung (gewöhnlicher) RMI-basierter Programme ist (neben den zahlreichen Veränderungen am Quelltext gegenüber dem lokalen Programm), daß Methodenaufrufe über RMI stets über Stub- und Skeletonklassen abgewickelt werden. Der dabei entstehende Overhead ist aber unnötig, wenn sich beide Objekte zufällig auf (physikalisch) derselben Maschine befinden. Nach Messungen in [32] ist ein Methodenaufruf über RMI etwa 4000 mal langsamer als ein "gewöhnlicher" Methodenaufruf, selbst wenn sich RMI-Client und Server auf derselben Maschine befinden. Eine Lösung dieses Problems wird in Abschnitt 6.4.4 vorgestellt.

6.1. Verfügbarkeit

Versionen für UNIX und Windows95 (die Unterschiede beschränken sich im wesentlichen auf das Archivformat) können von der JavaParty Web-Seite[2] in Karlsruhe bezogen werden. Der Distribution liegt neben den ausführbaren Dateien und einigen Beispielen auch der komplette Quelltext bei.

[1] `http://ParaStation.ira.uka.de/`
[2] `http://wwwipd.ira.uka.de/JavaParty/`

Um die ebenfalls verfügbaren effizienteren RMI- und Serialisierungsimplementierungen nutzen zu können, sind allerdings Modifikationen an den Java-Laufzeitbibliotheken erforderlich. Die entsprechenden Patches gibt es für die meisten Java-Versionen ebenfalls auf der genannten Web-Seite.

6.2. Erzeugung von Parallelität

JavaParty ermöglicht verteiltes Rechnen, indem es mehrere virtuelle Maschinen mit verteiltem Speicher zu einer VM mit gemeinsamen Speicher kombiniert. Mehrere Threads innerhalb einer virtuellen Maschinen laufen im Timesharing-Verfahren oder parallel, je nach Anzahl der Threads und Anzahl der Prozessoren der zugrundeliegenden Hardware. Threads auf verschiedenen JVMs kooperieren mittels RMI (vgl. Abschnitt 2.9).

Ein zentraler *Runtime Manager* übernimmt die Verwaltung der einzelnen VMs und die Verteilung der Objekte. Im Gegensatz zu anderen Lösungen unterstützt JavaParty eine automatische Objektmigration, um die Lokalität zu erhöhen und somit die Netzlast zu reduzieren. Zusätzliche Effizienz wird durch die eigene (optionale) RMI-Implementierung erreicht, die (geeignete Testmengen vorausgesetzt) eine Leistungssteigerung von bis zu 90% erreicht [29].

JavaParty ist ein Precompiler, der den Sprachumfang lediglich um das Schlüsselwort `remote` erweitert. Objekte, die als `remote` deklariert sind, werden in geeignete RMI-Klassen übersetzt, die auf entfernten Rechnern aufrufbar sind. Dadurch entlastet JavaParty den Programmierer vom Umgang mit den Besonderheiten, die die RMI-Programmierung mit sich bringt. Dazu werden vom Precompiler unter anderem folgende Veränderungen am Quelltext vorgenommen:

- Für jedes `remote`-Objekt muß eine Schnittstelle generiert werden, die alle zu verwendenden Methoden exportiert.

- Sowohl in der Schnittstelle, als auch in der implementierenden Klasse müssen diese Methoden so deklariert werden, daß sie eine `RemoteException` generieren. Dadurch müssen auch alle Aufrufe dieser Methoden im Quelltext verändert werden, um den neuen Ausnahmetyp behandeln zu können.

- Entfernte Klassen müssen von einer der RMI Basisklassen abgeleitet sein (üblicherweise von `UnicastRemoteObject`); da es in Java keine Mehrfachvererbung gibt, sind unter Umständen umfangreiche Veränderungen an der Klassenhierarchie notwendig.

Da es in Java zwar Zeiger, jedoch keine Adressen gibt, genügt es, gemeinsamen Speicher auf der Ebene von Objekten zu realisieren. Wenn es möglich ist, auf Membervariablen entfernter Objekte zuzugreifen und Methoden dieser Objekte aufzurufen, ohne daß dafür Änderungen am Quelltext vorgenommen werden müssen, dann verhält sich das Programm genau so, als ob sich alle Objekte in einem durchgehenden Adressraum befinden würden. Die Technik hierfür steht bereits im Java-Standard zur Verfügung, es handelt sich um die im Abschnitt 2.9 vorgestellte Remote Method Invocation (kurz RMI). Allerdings sind zur Transformation eines lokalen in ein verteiltes Programm deutliche Veränderungen am Quelltext notwendig.

JavaParty erleichtert dem Programmierer die Arbeit, indem es diese Modifikationen weitestgehend automatisiert, und verbessert zudem die Leistung gegenüber "gewöhnlichen" RMI-Programmen durch die Ausnutzung von Lokalität und die Möglichkeit, Objekte zwischen den einzelnen Maschinen zu verschieben. Um ein Java-Programm RMI-fähig zu machen, ist es notwendig, die Zugriffsbeschränkungen auf Variablen und Objekte aufzuheben, wodurch die Möglichkeiten des Compilers zur Fehlererkennung eingeschränkt werden. Da JavaParty als Precompiler realisiert ist, kann es diese Überprüfungen durchführen, bevor es selbst diese Zugriffsbeschränkungen aufhebt.

RMI beschränkt den Zugriff auf entfernte Objekte auf den Aufruf von Methoden; ein Zugriff auf Variablen ist nur möglich, wenn für jede Variable entsprechende Methoden `get` und `set` (wie sie für Java Beans üblich sind) implementiert werden. Diese Zugriffsmethoden werden von JavaParty bei Bedarf automatisch generiert, was den Aufwand bei der Programmierung deutlich verringert.

6.3. Synchronisation und Kommunikation

Die Synchronisationsmechanismen (`synchronized` Blöcke, die Methoden `wait` und `notify`, vgl. Abschnitt 2.6.2), die Java zur Verfügung stellt, sind auch in JavaParty-Programmen für lokale Objekte nutzbar, allerdings versagen diese im allgemeinen für entfernte Objekte [37]. Der Grund hierfür ist, daß ein entfernter Methodenaufruf technisch gesehen von zwei verschiedenen Threads behandelt wird einem auf der Maschine von der aus der Aufruf erfolgt und einem auf der Maschine auf der sich das aufgerufene Objekt befindet. Die Synchronisationsmechanismen in Java arbeiten jedoch streng Thread-bezogen. Das bedeutet, der Besitzer eines Monitors ist stets der Thread, der gerade Aktionen innerhalb dieses Monitors ausführt. Wird von diesem (solange er sich innerhalb des Monitors befindet) eine entfernte Methode aufgerufen, so übergibt er implizit die Kontrolle an einen anderen Thread, der allerdings nicht im Besitz des Monitors ist. Die Konsequenz ist in der Regel ein Deadlock, der im lokalen Fall nicht auftreten würde (vgl. Abbildung 6.1).

Um dieses Problem zu lösen, müßte das Konzept verteilter Threads eingeführt werden, die dann zur Laufzeit auf verschiedene lokale Threads abgebildet werden. Ein Thread ist in Java im wesentlichen eine Menge von Zustandsinformationen; dazu gehören Stackframes, Register und Informationen über zugeteilte Monitorobjekte. Die Verwaltung der Monitore ist aus Effizienzgründen zumeist in Maschinencode realisiert und bei der Verwaltung der Stackframes muß die Java-VM auf den Adressraum der ausführenden Maschine zurückgreifen. Aus diesen Gründen ist es für eine DSM- (Distributed Shared Memory) Implementierung, die als Precompiler realisiert und deutlich über der Adressebene angesiedelt ist, nicht möglich, transparent verteilte Threads zu realisieren. Zudem wäre hierfür wenigstens eine zentrale Verwaltung der Monitore (und damit verbunden auch der laufenden Threads) notwendig, was ohne Veränderung an der VM-Implementierung

```
remote class Deadlock
{
    synchronized meth1(boolean b)
    {
      // durch die automatische Objektverteilung befindet
      // sich das mit new erzeugte Objekt in der Regel nicht auf
      // derselben Maschine
        if (b)  (new Deadlock()).meth2(this);
    }

    synchronized meth2(XYZ x)
    {
      // dieser Aufruf blockiert jedoch, falls sich x auf einer
      // anderen Maschine befindet:
        x.meth1(false);
    }
}
```

Abbildung 6.1.: Deadlock im verteilten- harmlos im lokalen Fall

allerdings nur schwer zu realisieren ist.

JavaParty löst dieses Problem zumindest teilweise durch die Bereitstellung der Klasse `RemmoteTools`. Diese enthält äquivalente Methoden für `wait` und `notify`, die auch im verteilten Fall korrekt funktionieren. Zudem stellt es `RemoteThreads` als eigene Klasse zur Verfügung, die zwar nicht verteilt, aber wenigstens fernerzeugbar und migrierbar sind womit sie den meisten Anwendungen (inklusive verteilten Schleifen) genügen dürften.

6.4. Besonderheiten

Der JavaParty-Precompiler transformiert als `remote` deklarierte Klassen in geeignete RMI-Klassen, bevor das Programm mittels eines beliebigen Java-Compilers in Bytecode übersetzt wird. Abbildung 6.2 zeigt die einzelnen Schritte dieses Vorgangs. Obwohl JavaParty RMI als Backend nutzt und die generierten Klassen vollständig dem Java-Standard entsprechen, ist die Funktionalität im Gegensatz zur "gewöhnlichen" RMI-Programmierung erheblich erweitert.

Was zunächst wie ein Widerspruch erscheint, ist in Wirklichkeit nur eine Frage des Standpunkts. Natürlich läßt sich das Verhalten von JavaParty-Programmen durch geschickte Programmierung mit RMI simulieren. Jedoch sind die dafür notwendigen Veränderungen am Quellprogramm so erheblich, daß sie den Einsatz eines Precompilers nahelegen. JavaParty stellt genau diesen Precompiler zur Verfügung, der sich in den durchgeführten Tests als durchaus komfortabel erwies. Es stellte sich heraus, daß trotz des geringen Umfangs der Spracherweiterungen (ein einziges Schlüsselwort) die Ausdrucksstärke von Java, zusammen mit dem Wissen über die Arbeitsweise des Precompilers, völlig ausreichen um die Vorzüge von JavaParty sinnvoll zu nutzen.

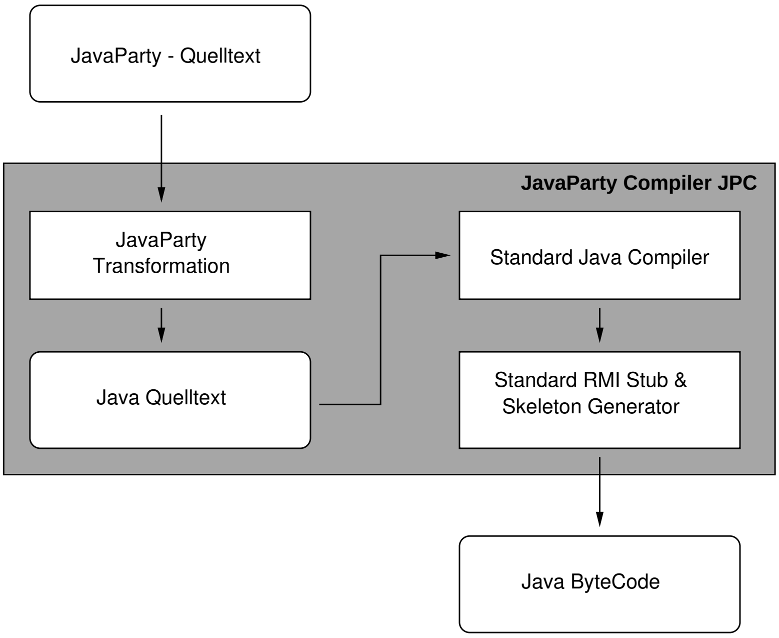

Abbildung 6.2.: Übersetzung von JavaParty-Programmen in Bytecode

Im folgenden werden die von JavaParty durchgeführten Transformationen näher untersucht um zu zeigen, wie sich dadurch einige der RMI-Besonderheiten umgehen lassen. Wie bereits in Abschnitt 2.9 deutlich wurde, bringt die RMI-Programmierung gewisse Enschränkungen mit sich, die die Arbeit mit entfernten Objekten behindern. Einige davon sind:

- Der Zugriff auf Klassenmethoden ist über RMI nicht möglich.
- RMI erlaubt keinen Zugriff auf (Member-)Variablen einer entfernten Klasse.
- Ein Zugriff auf RMI-Objekte erfolgt generell über die entsprechenden Stub- und Skeletonobjekte, selbst wenn sich das RMI-Objekt innerhalb derselben VM befindet.
- Es ist nicht möglich, RMI-Objekte von einem Rechner auf einen anderen zu verschieben.

6.4.1. Zugriff auf Klassenmethoden

Eine Klassenmethode ist in Java eine als `static` deklarierte Methode. Sie ist nicht, wie Objekt-Methoden, an eine Instanz des Objekts gebunden und hat deshalb auch nur auf Objektvariablen zugriff, die ebenfalls als `static` deklariert wurden; letztere werden gewöhnlich als Klassenvariablen bezeichnet. Die Deklaration einer Klasse läßt sich somit unterteilen in einen statischen Teil (oft auch Klassenteil genannt) und einen Instanzteil, welcher die Variablen und Methoden enthält, die später an eine Objektinstanz gebunden sein werden.

Abgesehen von Besonderheiten des Speichermanagements (vgl. Abschnitt 2.7) teilen sich alle Objekte einer Klasse eine gemeinsame Instanz jeder Klassenvariable. Die Veränderung einer Klassenvariable durch ein bestimmtes Objekt wird (früher oder später) für alle anderen Objekte sichtbar. Können verschiedene Threads zugreifen, müssen die Zugriffe in der Regel geeignet synchronisiert werden. Da die Java-Synchronisationsmechanismen stets an ein konkretes Objekt gebunden sind, muß für Klassenmethoden (und implizit auch -variablen) ein globales Synchronisationsobjekt erzeugt werden. In Java ist dieses Objekt stets eine Unterklasse von `java.lang.Class` und repräsentiert eine konkrete Klasse.

`B.class`	Proxyobjekt, das im generierten Programm anstelle von `B` verwendet wird. Es verbirgt den Aufruf der Stub- und Skeletonmethoden vor dem Benutzer.
`B_intf.class`	`interface` zum Zugriff auf Instanzvariablen und -methoden
`B_class.class`	`interface` zum Zugriff auf Klassenvariablen und -methoden
`B_impl.class`	Implementierung von `B_intf`
`B_class_impl.class`	Implementierung von `B_class`
`B_impl_Stub.class`	Vom RMI-Compiler erzeugte Stub-Klasse für `B_impl`
`B_impl_Skel.class`	Vom RMI-Compiler erzeugte Skeletonklasse für `B_impl`
`B_class_Stub.class`	Vom RMI-Compiler erzeugte Stub-Klasse für `B_class`
`B_class_Skel.class`	Vom RMI-Compiler erzeugte Skeletonklasse für `B_class`
`B.local`	Wird nur zur Übersetzung benötigt.

Tabelle 6.1.: Aus jeder `remote` Klasse `B` werden 10 neue Klassen generiert

Um dieses Verhalten im verteilten Fall zu simulieren ist ein gewisser Aufwand erforderlich. Abbildung 6.1 zeigt die Liste der Klassen, die von JavaParty für eine einzelne, als `remote` deklarierte Klasse `B` generiert werden. Dabei verschiebt der Precompiler, zunächst den Instanz- und den Klassenteil von `B` in getrennte Klassen, die später unterschiedlich referenziert werden können (vgl. Abbildung 6.3). Während die Referenzen auf den Instanzteil fast wie gewöhnliche Objektreferenzen behandelt werden können, muß die Laufzeitumgebung sicherstellen, daß für jede Klasse genau eine Instanz des Klassenteil-Objekts existiert.

Bei jeder Objektinstanziierung muß also zunächst eine lokale Version des Instanzteils erzeugt werden, anschließend muß diese geeignet mit einer RMI-Referenz auf den globalen Klassenteil verknüpft werden. Die Referenz auf den Klassenteil muß von der Laufzeitumgebung, also in der Regel selbst durch einen RMI-Aufruf, bezogen werden. Dieser Vorgang ist erheblich teurer als eine gewöhnliche Objektinstanziierung, was auch in den durchgeführten Tests schnell deutlich wurde. Abbildung 6.3 zeigt unter anderem den Bezug der Klassenteil-Instanz und den Aufruf von Klassenmethoden.

6.4.2. Zugriff auf Objektvariablen

Um den Zugriff auf die Variablen eines Objektes zu ermöglichen, genügt es prinzipiell, geeignete Methoden `get` und `set` für jede Objektvariable zu erzeugen. Um zu einer effizienten Lösung zu kommen, muß jedoch deutlich mehr Aufwand betrieben werden. Zusätzlich zu Lese- und Schreibmethoden müssen für numerische Variablen entsprechende Methoden für Pre- und Postfixincrement

```
a++;
++a;
```

erzeugt werden. Die kombinierten Zuweisungsoperatoren `+=`, `*=`, etc. lassen sich zwar auf einfache Zugriffsmethoden abbilden, bei Arrayzugriffen müssen jedoch Zugriffsmethoden für jede Dimension, sowie Zugriffs-, Increment- und Decrementmethoden für die einzelnen Elemente generiert werden. Ansonsten würde bei einem einfachen

```
int[] a = new int[64];
...

a[42]++;

...
```

das ganze Array zweimal übertragen um ein einzelnes Inkrement auszuführen. Da `Object` in Java der universelle Basistyp ist, lassen sich über Variablen vom Type `Object` theoretisch beliebig dimensionale Arrays ansprechen. Auch für diese müssen Zugriffsmethoden mit potentiell beliebig langen Indexvektoren erzeugt werden.

6.4.3. Parameterübergabe

Im Unterschied zu RMI erfolgt die Parameterübergabe bei JavaParty abhängig vom Speicherort des aufgerufenen Objektes (Tabelle 6.2). Während bei RMI gewöhnliche Objekte grundsätzlich per Wert an Methoden von Remote-Objekten übergeben werden, nutzt JavaParty vorhandene Lokalität und wählt die schnellere Übergabemethode per Referenz. Möglich wird dies durch die Verwendung von Proxyobjekten, die den Aufruf an geeignete Implementationsklassen weiterleiten (vgl. Abschnitt 6.4.4). Die sich daraus ergebende Semantik der Parameterübergabe wirkt zunächst schwer durchschaubar und fehlerträchtig, liefert aber in den meisten Fällen genau das gewünschte Verhalten.

	Aufgerufenes Objekt		
	“normal”	`remote`	`remote`
Parameter	lokal	lokal	entfernt
“normal”	Referenz	Referenz	**Kopie**
`remote`	Referenz	Referenz	Referenz

Tabelle 6.2.: Parameterübergabe an lokale und entfernte Methoden

Technisch gesehen werden alle Objekte bei einem entfernten Methodenaufruf per Kopie übergeben. Für als `remote` deklarierte Objekte werden anstelle der Objekte selbst die entsprechenden Stubobjekte übergeben, so daß die Semantik für diese Objekte einer Übergabe per Referenz entspricht, wie Tabelle 6.2 zeigt.

6.4.4. Ausnutzung von Lokalität

Da JavaParty RMI als Backend benutzt, kann der Zugriff auf Variablen oder Methoden eines entfernten (`remote`) Objekts nur so effizient sein, wie ein entsprechender Methodenaufruf in der zugrundeliegenden RMI-Implementierung. Wie sich aus der Beschreibung der Parameterübergabe über RMI in Abschnitt 2.9 unschwer erkennen läßt, ist der Aufruf einer Methode eines entfernten Objektes somit um Größenordnungen teurer als ein lokaler Methodenaufruf. Eine einfache Optimierung wäre es, vor dem Aufruf zu überprüfen, ob sich das aufzurufende Objekt auf dem lokalen Rechner befindet. In diesem Fall läßt sich der Overhead vermeiden, indem anstelle der Stub-Methode einfach direkt die Methode des eigentlichen Objektes aufgerufen wird. Dies ist in RMI leider nicht möglich, dennoch löst JavaParty dieses Problem und zwar auf folgende Weise.

An jeder Stelle, an der im Quellprogramm Bezug auf ein `remote` Objekt genommen wird, werden vom Precompiler Referenzen auf das entsprechende Proxyobjekt generiert. Das bedeutet: anstelle der Methode des ursprünglichen Objekts wird in der generierten Version des Programms stets die äquivalente Methode des Proxyobjekts aufgerufen[3].

Das Proxyobjekt enthält die notwendigen Referenzen, um beim Aufruf zwischen dem RMI-Methodenaufruf oder dem direkten Methodenaufruf zu wählen. Letzterer ist nur möglich, wenn sich das ursprüngliche Objekt in derselben virtuellen Maschine befindet, ist dann aber deutlich schneller, da das aufwendige Marshalling / Unmarshalling der Parameter entfällt.

6.4.5. Objektmigration

Ein weiterer Vorteil dieser Implementierung ist, daß es möglich ist, die Referenzen auf die Stubobjekte zur Laufzeit auszutauschen. Da in den Stubobjekten auch die Netzwerkadresse gespeichert ist, unter der das zugehörige Skeletonobjekt zu finden ist, ist es mit

[3]Dies gilt im übrigen auch für Zugriffe auf Variablen, die für die automatisch entsprechende Zugriffsmethoden generiert wurden.

RMI nicht möglich, entfernte Objekte von einem Rechner auf einen anderen zu verschieben. Um dies zu erreichen, müßten alle existierenden Stubobjekte dieses Objekts auf die neue Adresse angepaßt werden. Dies ist jedoch aufwendig und schwer zu realisieren, da normalerweise keine Rückwärtsreferenzen auf die Stubobjekte eines Objekts verwaltet werden.

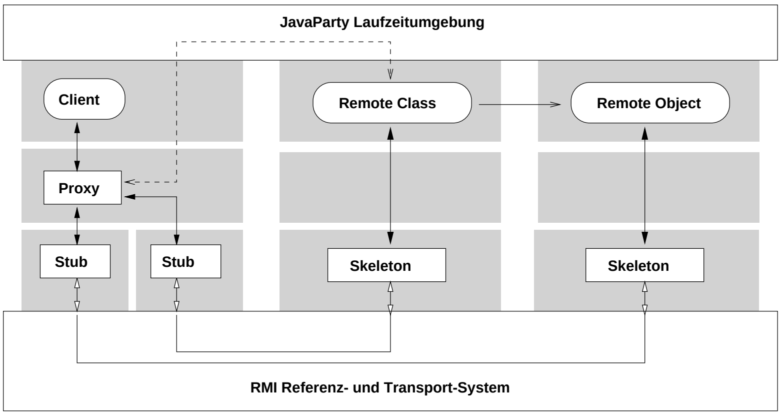

Abbildung 6.3.: Aufruf von Methoden entfernter Objekte über Proxies

In JavaParty lassen sich `remote`-Objekte dennoch verschieben. Die Lösung besticht durch ihre Pragmatik: Wird ein `remote` Objekt verschoben, so bleibt ein degradiertes Objekt zurück, welches ein aktuelles Stubobjekt mit der neuen Adresse des umgezogenen Objekts besitzt. Jeden Methodenaufruf quittiert es mit einer `MovedException` und übergibt damit das neue Stubobjekt an den Aufrufer. Der Aufrufer aktualisiert daraufhin seine Objektreferenz und startet den Aufruf erneut. Die Details hierzu finden sich in [37], woraus auch Abbildung 6.3 entnommen ist.

Ein Nachteil dieser Methode ist, daß bei jedem Umzug das ursprüngliche Objekt zurückbleibt, mit der Aufgabe, auf das Umzugsziel zu verweisen. Wird nun ein Objekt öfter zwischen zwei Rechnern verschoben, führt dies dazu, daß sich auf beiden Seiten veraltete Objekte ansammeln, die von der Speicherbereinigung nicht immer beseitigt werden können. Gibt es nun ein Proxyobjekt, welches nicht aktualisiert wurde (etwa weil lange Zeit keine Methode aufgerufen wurde, was zu einer `MovedException` führen würde), so kann keines der alten Objekte entlang der Verweiskette bis zum aktuellen Objekt abgeräumt werden [37].

7. Fallstudien

Zur praktischen Untersuchung der verschiedenen Projekte wurden drei ausgesuchte Algorithmen mit den verschiedenen Dialekten implementiert.

Als Testsystem für verteilte Programme diente ein Netzwerk von vier SUN-Workstations (3× 167MHz und 1× 330MHz), die durch eine 100MBit Ethernet Architektur verbunden sind. Als Java-Version diente jdk1.2.1 von SUN für Solaris. Für Untersuchungen auf einem Rechner wurde zusätzlich ein Intel-PC (PentiumII 400MHz) unter Linux (Kernel 2.2.13, glibc 2.1) benutzt. Auf diesem System stand die Java-Version 1.2.2 (RC3) von Blackdown[1] zur Verfügung.

Die Problemgrößen wurden so gewählt, daß die Laufzeit auf beiden Systemen etwa zwischen einigen Sekunden und einigen Minuten betrug. Das obere Ende dieses Bereichs wird im weiteren gelegentlich auch als "realistische" Problemgröße bezeichnet. Da es sich bei den Versuchsplattformen zum Teil um offene Systeme handelt, konnten äußere Einflüsse bei den Messungen nie ganz ausgeschlossen werden; die ermittelten Ergebnisse sind somit vor allem qualitativ zu interpretieren.

Jeder der ausgewählten Algorithmen wurde zunächst als sequenzielles Java-Programm und dann mit den zu untersuchenden Dialekten realisiert. Bei den Messungen verteilter Programme wurden stets zuerst die langsameren Arbeitsstationen (Versuchsreihen auf einer bis drei Maschinen) und zuletzt der etwas schnellere Rechner (Versuchsreihe auf vier Maschinen) belegt. Die Verteilung der Prozesse auf die einzelnen Rechner erfolgte zufällig. Die dabei ermittelten Resultate finden sich in den Abschnitten 7.1, 7.2 und 7.3.

Für die Projekte, in denen eine Skalierbarkeit aus technischen Gründen nicht untersucht werden konnte, diente die Laufzeit des reinen Java-Programms dann ersatzweise als Vergleichswert (Abschnitt 7.4). Eine Zusammenfassung der Ergebnisse findet sich in Abschnitt 7.5

Auswahl der Algorithmen Anhand von drei Beispielalgorithmen wurde versucht, einen möglichst umfassenden Eindruck über die Stärken und Schwächen der untersuchten Projekte zu gewinnen. Folgende Algorithmen wurden implementiert:

- LU-Zerlegung, ein numerischer Algorithmus mit festen Schleifengrenzen
- RT-Hülle, ein Graphalgorithmus mit dynamischen Schleifengrenzen
- Mergesort, ein Sortieralgorithmus nach dem Prinzip des "Divide and Conquer"

[1] `http://www.blackdown.org`

Während sich mit Titanium, HighPerformance und JavaParty annähernd alle Probleme sinnvoll lösen ließen, mußte auf eine Untersuchung mit HPJava leider verzichtet werden. An einer Fertigstellung des Frontends wird weiterhin gearbeitet [1]. Jedoch war bis zum Ende dieser Arbeit keine funktionsfähige Version zu bekommen. In der Zusammenfassung in Abschnitt 7.5 werden deshalb die wichtigsten (zu erwartenden) Eigenschaften von HPJava und ihre Eignung für die beobachteten Problemstellungen nur kurz vorgestellt.

LooPo *LooPo*[2] ist der Prototyp eines am Lehrstuhl für Programmierung der Universität Passau entwickelten Werkzeugs zur Analyse und modellbasierten Parallelisierung von Schleifenprogrammen. Die Details der dabei eingesetzten Verfahren finden sich in u.a. [20, 27] und sollen an dieser Stelle nicht weiter erläutert werden.

Für diese Arbeit stand die Fragestellung im Vordergrund, wie einfach sich mit LooPo erzeugte Zielprogramme mit den zu untersuchenden Sprachen umsetzen lassen. Aus diesem Grund ist der erste Abschnitt (Umsetzung der LU-Zerlegung) dieses Kapitels fast vollständig der automatischen Erzeugung der Zielprogramme gewidmet. Da die hierbei gewonnenen Erkenntnisse auf die beiden anderen Algorithmen übertragbar erschienen, wurde auf die Untersuchung LooPo-generierter Programme in den anderen Abschnitten verzichtet. Wie gesagt: nicht die Parallelisierungsfähigkeiten von LooPo sollten untersucht werden, sondern die Möglichkeiten der verschiedenen Java-Dialekte.

Die aktuelle Version von LooPo akzeptiert als Eingabe Schleifensätze in C oder FORTRAN Syntax; da die Syntax von C und Java (zumindest was Arrayzugriffe und Kontrollstrukturen betrifft) sehr ähnlich ist, war auch die Verarbeitung von Java-Schleifensätzen kein Problem. Die erzeugte Ausgabe läßt sich durch Syntaxdefinitions-Dateien flexibel an verschiedene Zielsprachen anpassen, wodurch zumeist nur wenig manuelle Nachbearbeitung erforderlich war.

Meßverfahren Die Zeiten wurden extern (d.h. von dem aufrufenden Prozeß aus) mit einem kleinen C-Programm gemessen. Bei diesem Programm handelt es sich um eine angepaßte Version des GNU `time` Programms, aus dessen Quellcode die Ermittlung von unrelevanten Meßwerten (Resourcenauslastung, CPU-Zeit, etc.) entfernt wurde. Die Zeitnahme erfolgt dabei über die UNIX `gettimeofday` Funktion, die auf den benutzten Plattformen eine Auflösung von etwa $\frac{1}{1000}$ Sekunde besitzt. Nach der ersten Zeitnahme wird mittels `fork` ein Kindprozeß gestartet, der das zu untersuchende Programm mittels `execvp` ausführt. Nachdem der Elternprozeß mittels `wait3` erfolgreich auf die Beendigung des Kindprozesses (und damit das Ende des zu untersuchenden Programms) gewartet hat, erfolgt die zweite Zeitnahme und die Ausgabe der Differenz.

Die so ermittelte Laufzeit eines Programms beinhaltet somit die gesamte Lebensdauer vom Aufruf bis zur Terminierung des Prozesses. Um störende Einflüsse herauszufiltern, wurden alle Messungen mehrmals (in der Regel dreimal) ausgeführt und aus den ermittelten Zeiten wurde der Durchschnitt berechnet.

[2] `http://www.fmi.uni-passau.de/loopo/`

7.1. LU-Zerlegung

Die LU-Zerlegung einer quadratischen nichtsingulären Matrix $A = (a_{ij})$, wobei $(i, j = 1, \cdots, n)$ liefert als Ergebnis eine obere und untere Dreiecksmatrix $U = (u_{ij})$ ("upper") und $L = (l_{ij})$ ("lower") mit $A = LU$. Die Hauptdiagonale von U besteht nur aus Einsen, ihre übrigen Werte sind entsprechend normiert. L und U sind wie folgt definiert[3]:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-i} l_{ik} u_{kj}, \quad j \leq i, i = 1, 2, \cdots, n \tag{7.1}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}}, \quad j > i, j = 2, \cdots, n \tag{7.2}$$

```
for (int i=0; i<n; i++) {
    for (int j=0; j<=i; j++) {
      double sum = A[i][j];

      for (int k=0; k<j; k++) {
        sum -= (L[i][k] * U[k][j]);
      }
      L[i][j] = sum;
    }

    for (int j=i; j<n; j++) {
      double sum = A[i][j];

      for (int k=0; k<i; k++) {
        sum -= (L[i][k] * U[k][j]);
      }
      U[i][j] = sum / L[i][i];
    }
}
```

Abbildung 7.1.: Java-Implementierung der LU-Zerlegung

Das Java-Programm in Abbildung 7.1, zeichnet sich durch seine nicht perfekt verschachtelten Schleifen aus. Die Schleifengrenzen sind ebenso wie die Indizes affin. Alle Abhängigkeiten sind entweder uniform oder ebenfalls affin. Für die Analyse der Abhängigkeiten sowie das Erzeugen einer parallelisierten Version stand eine aktuelle LooPo-Version zur Verfügung (siehe oben). Das von LooPo parallelisierte Programm wurde versuchsweise mit allen Projekten untersucht; zusätzlich wurde für JavaParty eine parallele Version von Hand erstellt.

[3]Eine genauere Untersuchung zur Parallelisierung dieses Beispiels findet sich in [16].

7.1.1. Implementierung mit Titanium

Titanium benötigt für eine verteilte Ausführung der Programme ein spezielles Backend, das auf der Split-C Bibliothek basiert. Diese Bibliothek (und somit auch das entsprechende Backend) waren für die Testplattform leider nicht verfügbar. Mit dem verfügbaren `pthread`-Backend lassen sich zwar parallele Programme erzeugen (und damit prinzipielle Eigenschaften wie Verklemmungsfreiheit und Synchronisation untersuchen), allerdings laufen diese Programme auf dem (Uniprozessor) Testsystem im Timesharing-Verfahren und erzielen somit keine Beschleunigung. Zudem lassen sich so die Kosten der impliziten Kommunikation und der globalen Synchronisation nicht ermitteln, die potentiell jedoch sehr hoch sind.

Da in Titanium alle Zeigertypen standardmäßig global sind, und die Referenzsemantik von Java für die Parameterübergabe beibehalten wurde (im Gegensatz etwa zu JavaParty oder RMI), ist der Kommunikationsbedarf vermutlich relativ hoch. Lokale Variablen[4] können vom Programmierer als solche gekennzeichnet werden, jedoch gestattete der Compiler nicht, ganze Titanium-Arrays als `local` zu deklarieren. Eine Lösung dieses Problems wäre, die Eingabearrays von Hand auf die beteiligten Rechner zu verteilen und selbst für eine geeignete (blockweise) Kommunikation zu sorgen.

Da sich Titaniumprogramme auf der Versuchsplattform jedoch ohnehin nicht verteilt ausführen ließen, wurde auf eine nähere Untersuchung verzichtet. Ein (möglicherweise übertragbarer) Vergleich zwischen impliziter und expliziter Kommunikation findet sich im Abschnitt 7.1.3 (Implementierung mit JavaParty). In diesem Abschnitt wird stattdessen zunächst die automatische Generierung von Titaniumprogrammen mit Hilfe von LooPo untersucht.

Automatische Erzeugung mit LooPo

Durch eine geeignete Ergänzung der Syntaxdefinitionsdatei von LooPo war es möglich, Zielprogramme für Titanium fast automatisch zu generieren. Dazu wurde die Definitionsdatei für C-Code an einigen Stellen verändert um der abweichenden Syntax für Titanium-Arrays und Synchronisationsstatements Rechnung zu tragen. Eine manuelle Nachbearbeitung des Zielprogramms war anschließend nur noch bei den Arraydeklarationen, sowie bei einigen Typkonvertierungen erforderlich.

Bei der Erzeugung des Zielcodes durch LooPo wurde versucht, den Besonderheiten von Titanium (SPMD, statisches Prozeßmodell, globale Synchronisation) gerecht zu werden. Die Implementierung berechnet jeden Zeitschritt asynchron mit explizitem Tiling, welches von LooPo unterstützt wird[5]. Nach jedem Zeitschritt erfolgt eine globale Synchronisation mittels eines `Ti.barrier` Methodenaufrufs. Die Allokation der benötigten Arrays erfolgt durch den Prozeß[6] mit ID 0. Von diesem werden die (globalen) Zeiger

[4] Der Ausdruck "lokal" in "lokale Variablen" bezieht sich hier nicht wie normalerweise üblich, auf die Sichtbarkeit, sondern auf den Speicherort, sprich: die lokale Maschine

[5] Die äußerste `parfor` Schleife wurde aus dem Zielprogramm entfernt, da Titanium beim Programmstart ohnehin die benötigten Prozesse erzeugt.

[6] Ähnlich wie bei MPI besitzt jeder Titanium-Prozeß eine eindeutige ID $i \in 0, \dots, N$, wobei N die Anzahl der Prozesse ist

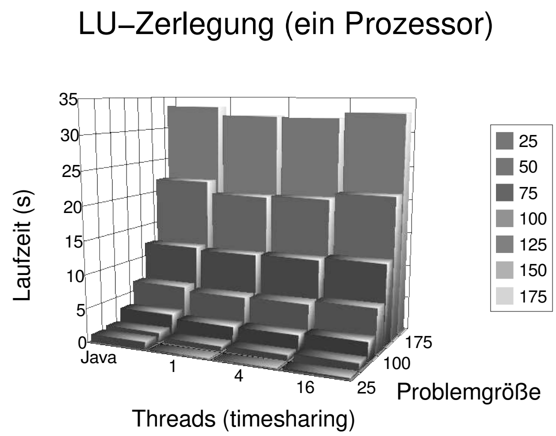

Abbildung 7.2.: Laufzeit der LU-Zerlegung mit Titanium

mittels `broadcast` an die übrigen Prozesse weitergegeben. Wie in Abschnitt 3.4.3 bereits erwähnt, bleiben die Daten dabei (physikalisch) in der Region des Prozeß mit der Nummer 0; für den Zugriff auf einzelne Array-Elemente durch andere Prozesse ist eine Kommunikation mit Prozeß 0 nötig. Diese Verteilung ist sicher nicht optimal, mangels einer verteilten Testumgebung wurde auf eine weitere Optimierung an dieser Stelle jedoch verzichtet.

Abbildung 7.2 zeigt die Laufzeit des Algorithmus auf der Linux-Plattform. Die Messungen wurden mit dem `pthread`-Backend auf einem Einprozessorsystem durchgeführt, die Werte unterscheiden sich jedoch nur im Rahmen der Meßungenauigkeiten von denen des `uniproc`-Backends.

Bemerkenswert ist dabei, daß das Titaniumprogramm eine etwas bessere Laufzeit erzielt, als das zum Vergleich angeführte Java-Programm. Bei diesem handelt es sich um ein ebenfalls von LooPo generiertes Zielprogramm, das mit einem standard Java-Compiler übersetzt und sequenziell ausgeführt wurde. Das Ergebnis überrascht, da die Titanium-Version auf Titanium-Arrays arbeitet, und diese, wie die Untersuchungen in Abschnitt 7.4.1 zeigen, in der Regel eine Verschlechterung der Laufzeit gegenüber reinen Java-Programmen verursachen.

Die Ursache hierfür ist möglicherweise, daß der von LooPo generierte Code relativ komplex ist und ein größeres Optimierungspotential besitzt, als die zumeist relativ kurzen, manuell erzeugten Schleifen der anderen Beispiele. Im Gegensatz zu konventionellen Java-Compilern führt Titanium eine Reihe von Optimierungen durch, die hier offensichtlich die Laufzeit soweit verringern, daß der erwartete Anstieg (durch die Verwendung der Titanium-Arrays) nicht mehr meßbar ist

Eine genauere Untersuchung zur Laufzeit von sequenziell ausgeführten LooPo-Programmen findet sich in Abschnitt 7.4.3. Festzustellen ist auch, daß Synchronisation und Taskwechsel die Laufzeit kaum beeinflussen, wie der geringe Anstieg der Laufzeit beim Übergang von einem auf 16 Threads beweist.

7.1.2. Implementierung mit HighPerformanceJava

Die Erzeugung eines Zielprogramms für HighPerformanceJava scheiterte an verschiedenen Faktoren. Obwohl es prinzipiell möglich ist, sind die Anforderungen an die Codegenerierung höher, als sie im Rahmen dieser Arbeit sinnvoll untersucht werden können. Hinzu kommt, daß die so erzeugten Programme nur noch sehr wenig von der Funktionalität von HighPerformanceJava nutzen. Dieser Abschnitt beschreibt deshalb nur die entstandenen Probleme und macht zudem einige Vorschläge, wie sich diese unter Umständen lösen lassen.

Das HighPerformanceJava-Zielprogramm sollte sich eigentlich mit wenig Aufwand automatisch erzeugen lassen: Es gibt ein Sprachkonstrukt für parallele Schleifen, mit dem sich die äußere Schleife eines asynchronen Zielprogramms einfach realisieren läßt. Ein explizites Tiling ist im Gegensatz zur Titanium-Version nicht notwendig, da dies genau die Funktionalität des Iterationspools ist, den HighPerformanceJava für jede parallele Schleife erzeugt.

Da kein Sprachkonstrukt für eine globale Synchronisation vorhanden ist, müßte lediglich die benötigte Barriersynchronisation geeignet simuliert werden. Genau dies erweist sich jedoch als relativ schwer, da HighPerformanceJava keine Methodenaufrufe innerhalb paralleler Schleifen erlaubt. Eine globale Synchronisation, die ohne `wait` und `notify` Aufrufe auskommt konnte leider nicht realisiert werden.

Eine andere Möglichkeit, die dieses Problem umgeht, ist die Erzeugung eines synchronen Zielprogramms. Da wegen des dynamischen Threadmodells jede parallele Schleife in HighPerformanceJava einem `fork`/`join` entspricht, findet implizit eine globale Synchronisation statt. Da der von LooPo erzeugte Code eine Vielzahl von mathematischen Funktionen (`ceil`, `floor`, `max`, etc.) enthält, ist auch diese Version nicht ohne umfangreiche Nachbearbeitung ausführbar. Hinzu kommt, daß das Konzept des synchronen Programms der Task-Farming Idee von HighPerformanceJava nicht wirklich entspricht.

Zu diesen Schwierigkeiten kommt die Tatsache, daß `javar`, bei dem es sich um einen Prototyp handelt, einige Stabilitätsprobleme[7] bei der Parallelisierung der mit LooPo erzeugten Programme zeigt. Dies dürfte unter anderem an der durchgeführten Abhängigkeitsprüfung und der doch reichlich komplexen Struktur der erzeugten Schleifen liegen. Die Summe dieser Effekte verhinderte schließlich eine erfolgreiche Umsetzung.

7.1.3. Implementierung mit JavaParty

Auch eine JavaParty-Version ließ sich mit geringem Aufwand mit LooPo erzeugen, wie der erste Teil dieses Abschnitts beweist. Da sich JavaParty-Programme verteilt ausführen lassen, und somit in der Lage sind, auf den zur Verfügung stehenden Rechnern echte Beschleunigung zu erzielen, schien es angebracht, im Vergleich die Laufzeit eines manuell erstellten parallelen Programms zu untersuchen.

[7] In einigen Fällen, die nicht näher eingegrenzt werden konnten, terminiert `javar` nicht und erzeugt somit auch keine Ausgabe.

Automatische Erzeugung mit LooPo

Mit der (wiederum nur leicht veränderten) Syntaxdefinitionsdatei für C, lies sich auch das Zielprogramm für JavaParty weitgehend automatisch erzeugen. Eine manuelle Nachbearbeitung war vor allem bei den Variablendeklarationen erforderlich, da JavaParty weder verteilte noch globale Arrays unterstützt.

Um LooPo-generierte Programme mit JavaParty ausführen zu können, waren zudem einige zusätzliche Klassen nötig, die im folgenden kurz beschrieben werden. Diese Ergänzungen sind weitgehend unabhängig von dem untersuchten Algorithmus.

- Zum Verteilen der Berechnung wird ein `parfor` benötigt, welches in Form eines `fork`/`join` realisiert wurde. Dazu werden zu Beginn der Berechnung in einer Schleife die benötigten Threads erzeugt und gestartet (`fork`). Am Ende der Berechnung wird (wiederum in einer Schleife) auf die Terminierung jedes der zuvor gestarteten Threads gewartet (`join`). Beide Operationen wurden als Methoden in einer entsprechenden Klasse realisiert.

- Zur globalen Synchronisation wird eine `barrier`-Anweisung benötigt, die ebenfalls in einer zentralen Klasse realisiert wurde. Abbildung 7.4 zeigt die Implementierung nach einem Vorschlag von [23].

- Da es nicht möglich ist, Arrays oder Skalare als `remote` zu deklarieren, müssen diese in ein umgebendes `remote` Objekt "verpackt" werden. Diese Objekte enthalten die entsprechenden Arrays oder Skalare als Membervariablen und sind selbst als `remote` deklariert.

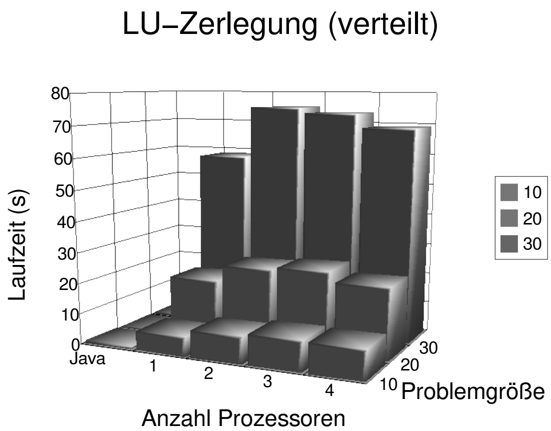

Abbildung 7.3.: Die Laufzeit der LU-Zerlegung auf dem Testsystem

Mit diesen Ergänzungen lassen sich LooPo generierte Programme zwar verteilt ausführen, eine Beschleunigung ist jedoch nicht zu erwarten. Der Grund liegt in der impliziten Kommunikation, die bei dem untersuchten Programm wenigstens eine Kommunikation pro Arrayzugriff erfordert. Die DSM-Simulation entspricht im wesentlichen der, die auch in der Titanium-Version benutzt wird (vgl. Abschnitt 7.1.1): die Daten liegen lokal auf einem Rechner, während die übrigen Maschinen nur Zeiger auf sie besitzen.

```
public void barrier() {
  synchronized (sem) {
    sem.counter -= 1;
    if (sem.counter > 0) {
        try {
          sem.wait();
        } catch (InterruptedException e) {
        }
    } else {
      sem.counter = jp.lang.DistributedRuntime.machines();
      sem.notifyAll();
    }
  }
}
```

Abbildung 7.4.: Eine Barriere mit JavaParty

Abbildung 7.3 zeigt das Ergebnis der Laufzeitmessungen. Die linke Spalte zeigt die Laufzeit des sequenziellen Java-Programms, die sich in diesem Maßstab allerdings kaum erkennen läßt. Die im Vergleich geringere Laufzeit auf einem Prozessor läßt sich auf die in Abschnitt 6.4.4 vorgestellten Zugriffsoptimierungen zurückführen. Die ansonsten annähernd gleiche Laufzeit zeigt, wie hoch die Kosten für Arrayzugriffe und Synchronisation im Vergleich zur eigentlichen Berechnung sind. Die sich abzeichnende Beschleunigung beim Übergang von zwei auf vier Rechner liegt angesichts der geringen Problemgröße (30×30) demnach auch eher im Bereich der Meßfehler.

Eine mögliche Verbesserung der Laufzeit könnte unter Umständen durch den Einsatz der ebenfalls in Karlsruhe entwickelten effizienten RMI-Implementierung erreicht werden. Ob diese ausreicht, um die Laufzeit bis in einen realistischen Bereich zu senken, ist jedoch fraglich. Eine alternative Lösung speziell für diesen kommunikationsintensiven Algorithmus wird daher im folgenden Abschnitt präsentiert.

Alternative Implementierung mit Kommunikation über Sockets

In der zweiten Version des Algorithmus wurde sowohl die Verteilung der Daten, als auch die Kommunikation explizit angegeben. Durch blockweise Kommunikation und geeignete Wahl der netzwerkspezifischen Puffergrößen erreicht diese Version wenigstens die Leistung des sequenziellen Java-Programms. Daß diese dabei nur knapp übertroffen wird, zeigt, wie schwer es ist, die Kommunikationskosten für diesen Algorithmus zu reduzieren.

Zur Berechnung der Ausgabematrizen L und U wird zunächst die Eingabematrix A vertikal in M (Anzahl der Maschinen) Teile zerlegt, und an die einzelnen Maschinen $m_1, \dots, m_M$ verteilt (Abbildung 7.5). Jede Maschine m_k $(1 \leq k \leq M)$ berechnet anschließend L_k und U_k wobei $L_k = (l_{ij})$, $(i = \theta(k), \dots, \omega(k))$, $(j = 1, \dots, n)$ und $U_k = (u_{ij})$, $(i = \theta(k), \dots, \omega(k))$, $(j = 1, \dots, n)$. $\theta(k)$ und $\omega(k)$ bezeichnen den kleinsten bzw. größten Index i, der von der Maschine m_k berechnet wird. Die Berechnung erfolgt für jeden Abschnitt zeilenweise. Bezogen auf Abbildung 7.1 entspricht dies einer Parallelisierung der beiden `j`-Schleifen.

Zwischen zwei Maschinen m_k und m_{k+1} besteht eine statische Socketverbindung. In diesen Socket wird jede berechnete Zeile elementweise geschrieben und so zur nächsten Maschine übertragen. Der Algorithmus sorgt dabei für die richtige Reihenfolge der verschickten `double` Werte; zusätzliche Prüf- oder Protokolldaten werden nicht übertragen. Die Socketimplementierung von Java erlaubt eine implizite Pufferung und ermöglicht so auf der einen Seite (bis zum Erreichen der Puffergröße) nicht blockierendes Schreiben und auf der anderen Seite (bis die benötigten Ergebnisse übertragen wurden) blockierendes Lesen. Auf diese Art ist keine zusätzliche Synchronisation erforderlich; die Übertragung erfolgt blockweise mit einer wählbaren Puffergröße und somit (hoffentlich) mit größtmöglichem Durchsatz.

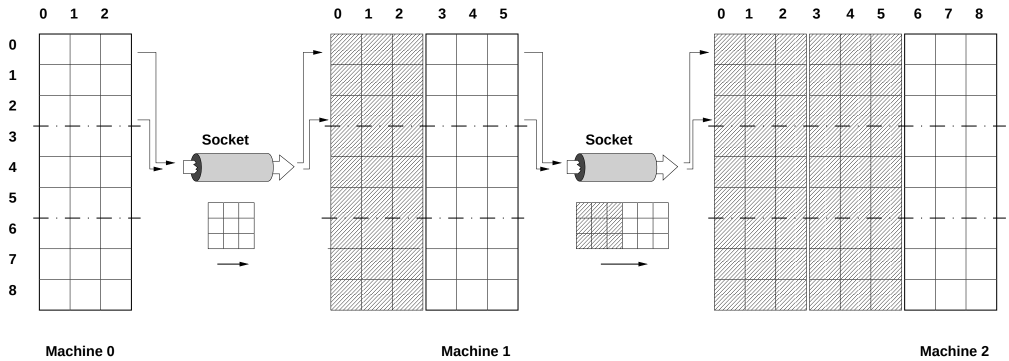

Abbildung 7.5.: Verteilung der Berechnungen bei der LU-Zerlegung

Abbildung 7.5 zeigt die Berechnung einer 9×9 Matrix durch drei Maschinen. Der dargestellte Indexraum bezieht sich gleichermaßen auf L und U. Die grau hinterlegten Felder sind Zwischenergebnisse, die von anderen Maschinen übertragen wurden. Die weißen Flächen symbolisieren Indizes, die lokal berechnet werden.

Da zur Berechnung von l_{ij} und u_{ij} jeweils a_{ij}, sowie $l_{i'j}$, $u_{i'j}$, $l_{ij'}$ und $u_{ij'}$ $(1 \leq i' < i$, $1 \leq j' < j)$ benötigt werden, werden von jeder Maschine auch die erhaltenen Zwischenergebnisse (d.h. für jede Zeile das Stück vom Anfang bis zum ersten selbst berechneten Index) an die jeweils nächste Maschine übertragen. Zuletzt wird das Ergebnis von der Maschine m_M zurückübertragen. Daraus ergibt sich ein geschätzter Kommunikationsbedarf von

$$V = \left(1 + \frac{(M-1)}{2}\right) Nc$$

wobei V das Datenvolumen und c einen konstanten Faktor bezeichnet, in dem unter anderem die Größe der einzelnen Datenelemente berücksichtigt ist. $N = n^2$ ist die

Anzahl[8] der zu übertragenden Elemente für L und U und $M > 1$ ist die Anzahl der benutzten Maschinen.

Abbildung 7.6 zeigt die Laufzeit des Algorithmus auf der Testplattform. In der linken Spalte ist wiederum die Laufzeit des sequenziellen Java-Programms auf einem Rechner dargestellt; die übrigen Spalten zeigen die Laufzeit des parallelen Programms auf unterschiedlich vielen Rechnern. An dieser Stelle sei auf die (im Vergleich zum letzten Beispiel) deutlich größere Problemgröße (525×525 anstelle von 30×30) bei annähernd halbierter Laufzeit hingewiesen. Deutlich zu erkennen ist der lineare Anstieg der Laufzeiten für kleine Problemgrößen. Dieser ist in erster Linie auf den Overhead zurückzuführen, der durch die entfernte Objekterzeugung und den Aufbau der Socketverbindungen entsteht.

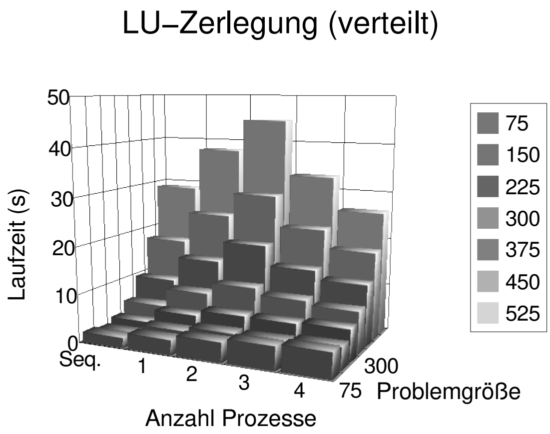

Abbildung 7.6.: Die Laufzeit der LU-Zerlegung auf dem Testsystem

Der deutliche Anstieg beim Übergang von einem auf zwei Rechner zeigt, wie hoch die Kommunikationsaufwand selbst in diesem Beispiel noch ist. Erst mit drei Rechnern erreicht der parallele Algorithmus eine bessere Laufzeit als auf einem Rechner, liegt jedoch immer noch deutlich hinter der sequenziellen Version; erst ab vier Rechnern wird eine echte Beschleunigung gegenüber dem sequenziellen Programm erreicht. Da die untersuchte Implementierung bereits die unter Java effizienteste Kommunikation (statisch verbundene, gepufferte Sockets ohne zusätzlichen Protokoll-Aufwand) verwendet, verspricht auch der für JavaParty verfügbare effiziente RMI-Ersatz keine Abhilfe. Möglicherweise lassen sich die Kommunikationskosten mit einer für das in Karlsruhe entwickelte ParaStation Netzwerk angepaßten JavaParty-Version noch senken. Dies konnte im Rahmen dieser Arbeit jedoch nicht weiter untersucht werden.

Anmerkungen und mögliche Verbesserungen Obwohl die in diesem Abschnitt präsentierte Version vollständig manuell erstellt wurde, sollte es prinzipiell möglich sein, eine

[8]Die Hauptdiagonale von U besteht per Definition nur aus Einsen und muß deshalb nicht übertragen werden, ebensowenig die obere bzw untere Dreiecksmatrix von L und U, da diese nur Nullen enthalten.

ähnliche Version auch mit LooPo zu erzeugen. Dazu müßten zunächst explizite Kommunikationsbefehle generiert und diese anschließend geeignet nachbearbeitet werden, um von der impliziten Pufferung der Socketkommunikation in Java zu profitieren. Eine bessere (und vor allem kontrolliertere) Ausnutzung der vorhandenen Parallelität sollte sich erreichen lassen, wenn es gelingt, die Kommunikation bereits auf Programmebene blockweise abzuwickeln. In diesem Fall kann ein Schreiben der Daten mittels `OutputStream.flush` nach jedem Block erzwungen werden. Gerade für Algorithmen mit unregelmäßigem Kommunikationsaufkommen kann so ein unnötiges Warten des Empfängers (wie es bei einer implizit gepufferten Übertragung vorkommen kann) verhindert werden.

7.2. RT-Hülle

Zu bestimmen ist die *reflexiv-transitive Hülle* eines Graphen. Der Graph ist dabei als eine Menge von Adjazenzlisten gegeben; für jeden Knoten ist in einem Array die Menge seiner Nachfolgeknoten abgebildet. Der Algorithmus berechnet die reflexiv-transitive Hülle für jeden Punkt wiederum als Array, indem er sukzessive alle erreichbaren Knoten hinzufügt.

Das Verfahren zeichnet sich durch die Verwendung von verschachtelten Schleifen aus, wobei die Schleifengrenzen erst zur Laufzeit bekannt sind. Der Algorithmus arbeitet im Grunde auf verketteten Listen, die aus Effizienzgründen jedoch als Arrays realisiert sind. Um den Kern des Problems dadurch nicht zu verzerren, verzichtet die Implementierung auf eine Abfrage der Schleifengrenzen (wie sie in Java möglich ist), und markiert statt dessen das Ende jeder Liste mit dem Eintrag $\perp$ (Bottom) im darauffolgenden Arrayelement.

Die allgemeine Problematik der Parallelisierung von Schleifen, deren Grenzen sich zur Laufzeit dynamisch ändern, ist nicht Thema dieser Arbeit. Die Details dazu finden sich in [19], von wo auch dieses Beispiel entnommen ist. Hier soll nur die Eignung der verschiedenen Java-Dialekte für die Implementierung des bereits parallelisierten Programms untersucht werden.

Abbildung 7.7 zeigt eine sequenzielle Java-Implementierung des Algorithmus. Die Semantik der `for`-Schleifen in Java entspricht (ähnlich wie in C) der von `while`-Schleifen und liefert so die dynamischen Schleifengrenzen, wie sie für den Algorithmus benötigt werden. Die Array-Darstellung des Graphen wird in Tabelle 7.1 genauer erklärt. Abbildung 7.8 zeigt einen einfachen Beispielgraphen; seine Darstellung in `succ` und `nrsuc` wird durch Abbildung 7.9 verdeutlicht.

Aus Effizienzgrüden wird in diesem Beispiel nur die äußere Schleife parallelisiert. Ihre obere Grenze entspricht der Anzahl der Knoten, die (wie bereits erwähnt) nicht explizit angegeben ist. Stattdessen werden ungültige Elemente mit $\perp$ markiert. Somit kann `nrsucc[n+1]==BOTTOM` als Abbruchkriterium benutzt werden. Dadurch entsteht allerdings eine Abhängigkeit $n \leftarrow n-1$ für $0 < n < N$. Die Details der Verteilung finden sich in den Unterabschnitten zu den einzelnen Projekten.

Eingabe-Arrays	
`succ`	Die Nachfolger des entsprechenden Knotens als zweidimensionales Array. Nur die Einträge `succ[i][j]` für $i < anzahlKnoten$, $j < nrsuc[i]$ werden benutzt. Alle anderen sind mit `BOTTOM` markiert.
`nrsuc`	Die Anzahl der Nachfolger des Knotens. Nichtbenötigte Einträge sind ebenfalls mit `BOTTOM` markiert.
Ausgabe-Arrays	
`rt`	Die reflexiv transitive Hülle des Knotens als zweidimensionales Array. Die erste Dimension entspricht wie bei `succ` der Nummer des Knotens. Die zweite Dimension entspricht der Nummer des Nachfolgers. Nicht benutzte Einträge sind auch hier mit `BOTTOM` markiert.
`tag`	Ist ein Hilfsarray, welches die bereits besuchten Knoten markiert.

Tabelle 7.1.: Die Datenstruktur für RT-Hülle

```
for (int n=0; nrsuc[n]!=BOTTOM; n++) {
   rt[n][0] = n;
   tag[n][0] = true;
   nxt[n] = 1;
   for (int d=0; rt[n][d] != BOTTOM; d++ ) {
      for (int s=0; succ[rt[n][d]][s]!=BOTTOM; s++) {
         if ( ! tag[n][ succ[rt[n][d]][s] ] ) {
            tag[n][ succ[rt[n][d]][s] ] = true;
            rt[n][nxt[n]] = succ[rt[n][d]][s];
            nxt[n]++;
         }

      }
   }
}
```

Abbildung 7.7.: Das Java-Quellprogramm für RT-Hülle

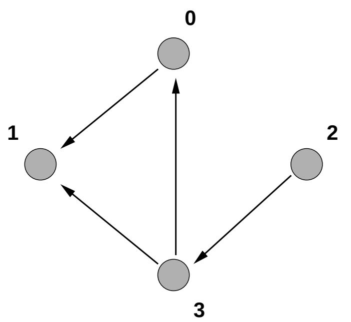

Abbildung 7.8.: Ein einfacher Beispielgraph

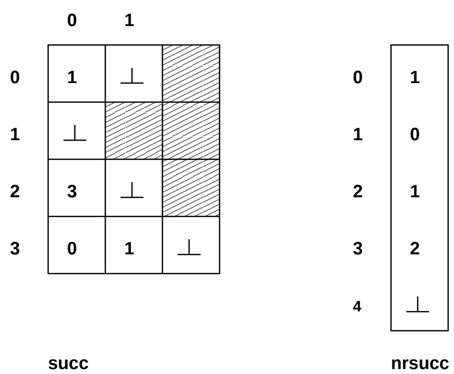

Abbildung 7.9.: Darstellung des Eingabegraphen in zwei Arrays `succ` und `nrsuc`

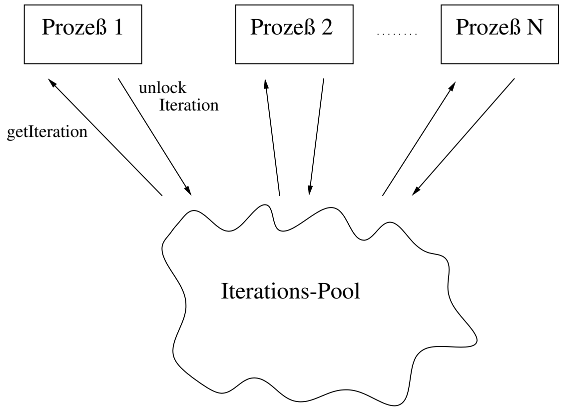

Abbildung 7.10.: Die Idee des Iterationspools

7.2.1. Implementierung mit Titanium

Als besonders praktikabel erscheint die Implementierungsidee von HighPerformanceJava implementiert mit Titanium. Die Verwendung eines Iterationspools garantiert bestmögliche Parallelität und vermeidet globale Synchronisation. Titanium bietet die notwendigen Sprachkonstrukte um diese Lösung einfach und (möglicherweise) effizient umzusetzen.

```
public synchronized Iteration getIteration() {
   while (stack.empty()) {
      if (!running) {
         notify();
         return null;
      } else {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
   }
   return (Iteration)stack.pop();
}
```

Abbildung 7.11.: Die Implementierung des Iterationspools mit Titanium

Abbildung 7.11 zeigt die Methode `getIteration`[9] der Titanium-Version des Iterationspools. Die Synchronisation über Monitore erfolgt wie in Java mittels als

[9] Die verfügbaren Iterationen werden hier als Stack verwaltet. Dieses Implementierungsdetail ist für die Funktionsweise des Algorithmus jedoch ohne Bedeutung.

`synchronized` markierter Methoden (vgl. Abschnitt 2.6.2). Titanium gestattet, eine Referenz auf das Pool-Objekt per Broadcast an alle Prozesse zu verteilen. Jeder Prozeß entnimmt eine Iteration aus dem Pool, fügt über eine `unlockIteration`-Methode neue Iterationen dem Pool hinzu und führt die eben entnommene Iteration aus. Solange keine Abbruchbedingung erreicht ist, enthält die `running` Marke des Pools den Wert `true`. In diesem Zustand werden Prozesse blockiert (`wait`, vgl. Abschnitt 2.6.1), die versuchen Iterationen aus einem leeren Pool zu entnehmen. Wenn der Pool später wieder Elemente enthält, werden diese Prozesse geweckt (`notify`) und können die Entnahme fortsetzen.

7.2.2. Implementierung mit HighPerformanceJava

Eine Implementierung mit HighPerformanceJava war leider nicht möglich. Obwohl diese Sprache im Prinzip alle Mittel zur Verfügung stellt, die zur Lösung von dynamischen Problemen geeignet erscheinen, verhindern in der Regel Randbedingungen und Implementierungsdetails deren Einsatz. Dieser Abschnitt beschreibt kurz die erfolglosen Versuche und die Gründe ihres Fehlschlagens.

Zunächst bereiten die dynamischen Schleifengrenzen Probleme bei der Implementierung. Unglücklicherweise gestattet `javar` nur parallele Schleifen mit konstanten Grenzen und ignoriert zudem parallele Schleifen innerhalb paralleler Schleifen. Obwohl HighPerformanceJava also auf der Implementierungsseite genau den Iterationspool zur Verfügung stellt, der für dieses dynamische Problem benötigt wird, verhindern die Einschränkungen des Precompilers seinen Einsatz. Was bleibt, ist eine Parallelschleife, die dazu benutzt werden kann, die Berechnung auf alle Prozessoren zu verteilen. So angewandt enthält der Iterationspool anfangs genau N (Anzhahl der Threads) Iterationen, von denen jeder Thread genau eine ausführt.

Das nächste Problem ist eine geeignete Synchronisation der so erzeugten Threads. Zwar stellt HighPerformanceJava zu diesem Zweck Vektoren von Synchronisationsbits bereit, jedoch unterliegen diese zwei wesentlichen Einschränkungen:

1. Jedes Bit kann nur einmal gesetzt, dann aber nicht mehr zurückgesetzt werden.
2. Ein wahlfreier Zugriff auf die Bits des Vektors ist nur beim Test möglich; gesetzt wir immer das Bit, dessen Index der aktuellen Schleifeniteration entspricht.

Zwar ist es möglich, beliebig viele dieser Vektoren für eine einzelne Schleife zu erzeugen, jedoch muß diese Zahl fest im Quelltext angegeben werden. Da in dem untersuchten Beispiel die Anzahl der Knoten im Graph beim Programmstart unbekannt ist, werden potentiell unendlich viele Synchronisationsbits benötigt. Das Problem könnte umgangen werden, liese sich explizit eine willkürliche Obergrenze für die Anzahl der benötigten Synchronisationsbits angeben. Eine universellere Lösung wäre es, eine leicht veränderte Version dieser Vektoren einzusetzen, die ein Rücksetzen der Bits erlauben. Die benötigte Operation entspricht einem atomaren "Test-And-Set", welches mit den zur Verfügung stehenden Mitteln realisierbar ist. Wie Abbildung 7.12 zeigt, ist der so erreichbare Grad an Parallelität jedoch stark von der Struktur des Graphen abhängig.

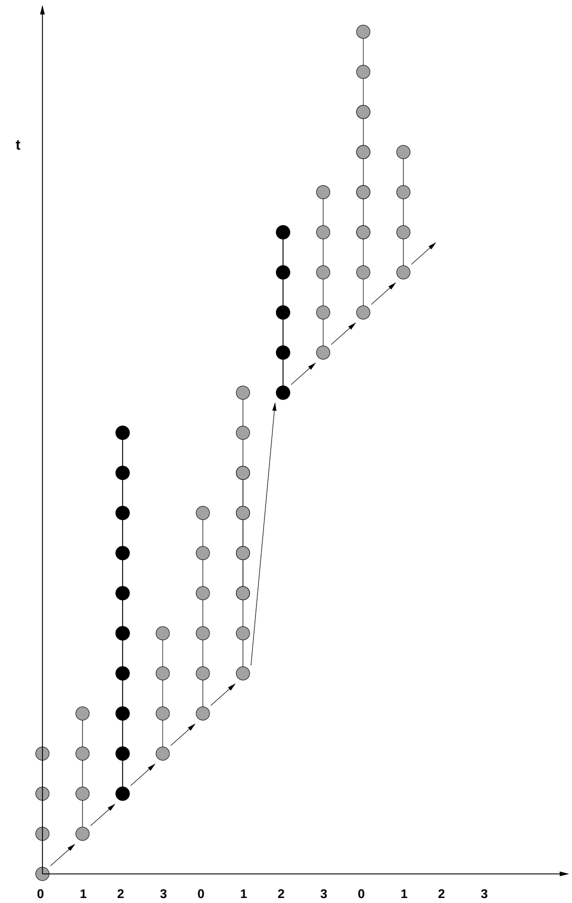

Abbildung 7.12.: HighPerformanceJava-Version von RT-Hülle

Da zudem diese Implementierung nahezu unverändert auch als reine Java-Version lauffähig wäre, wird sie in diesem Kapitel nicht weiter untersucht. Ziel der Arbeit ist es vielmehr, die Ausdrückbarkeit der betrachteten Probleme mit den besonderen Mitteln der betrachteten Sprache zu untersuchen.

7.2.3. Implementierung mit JavaParty

Die Java-Party-Implementierung unterscheidet sich nur geringfügig von der reinen Java-Version. Zunächst wird ein Objekt `RT` erzeugt und der Eingabegraph eingelesen. Die hierbei erzeugten Datenstrukturen (die Arrays `succ` und `nrsuc`) sind Instanzvariablen dieses Objektes. In der Methode `distribute` werden diese Daten auf die einzelnen Prozesse verteilt und die Berechnung gestartet.

An dieser Stelle erweist sich die Semantik der Parameterübergabe in JavaParty als durchaus angenehm zur Implementierung dieses Vorgangs. Da "normale" (also nicht als `remote` deklarierte) Objekte per Kopie an entfernte Objekte übergeben werden, reicht es aus, auf jeder der N beteiligten Maschinen ein `remote`-Objekt `Worker` zu erzeugen und dem Konstruktor die bereits vorhandene Klasse `RT` als Parameter zu übergeben. Die Erzeugung eines entfernten Objektes ist relativ unkompliziert:

```
jp.lang.DistributedRuntime.setTarget(machine);
new Worker(this,low,high,resultListener).start();
```

Unter den Parametern befinden sich die Unter- und Obergrenze der zu berechnenden Indizes (`low` und `high`), das `RT` Objekt (hier: `this`), welches als Kopie übergeben wird, und ein Zeiger auf den `ResultListener`, der als `remote` deklariert wurde und deshalb per Referenz übergeben wird. Auf diese Art werden N Instanzen der Eingabedaten und eine Instanz des `ResultListener`s erzeugt und verteilt. Die Implementierung macht sich die in Abschnitt 2.5 erwähnte Eigenart zunutze, daß Java-Arrays unterschiedlich lange Zeilen besitzen dürfen. Da stets ganze Objekte übertragen werden, läßt sich auf diese Art sehr einfach Speicherplatz und somit Bandbreite einsparen.

Da `Worker` von `jp.lang.RemoteThread` abgeleitet ist, wird mit dem Objekt auch gleich ein neuer Thread auf dem entfernten Rechner erzeugt und gestartet. Der Aufruf der Methode `start` kehrt sofort zurück, obwohl die Bearbeitung auf dem anderen Rechner gerade erst begonnen hat. Sobald die Berechnung terminiert, wird eine entsprechende Methode des ebenfalls übergebenen ResultListeners aufgerufen.

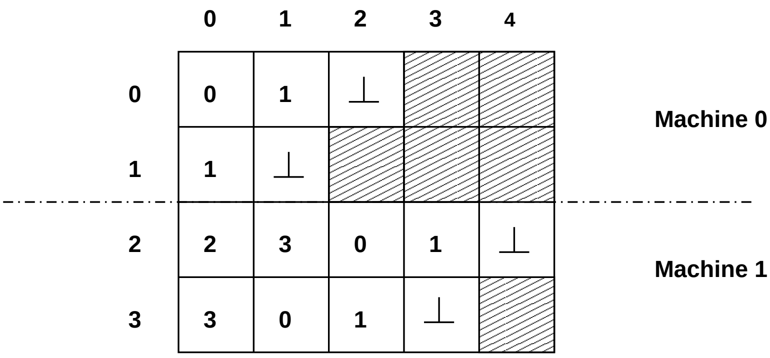

Abbildung 7.13.: Berechnung der Ausgabedaten durch zwei Maschinen

Diese Implementierung stellt einen Sonderfall des Iterationspools dar: jedes übergebene Objekt entspricht einer festen Anzahl von Iterationen; werden mehr dieser Objekte erzeugt, als Maschinen zu Ausführung bereit stehen, werden auf einer Maschine unter Umständen mehrere Iterationen im Timesharing berechnet. Dabei übernimmt der Scheduler des Betriebssystems in gewisser Weise die Auswahl der Iterationen aus dem Pool. Da die Kosten für die zusätzlichen Taskwechsel im Vergleich zu den Kommunikationskosten gering ausfallen, ist diese Art der Implementierung akzeptabel. Sie bietet zudem den Vorteil, daß in hybriden Netzen, in denen SMP- und Einprozessorsysteme kooperieren, auf den SMP-Rechnern so zusätzliche Parallelität erreicht wird.

Die Zuteilung der so erzeugten Objekte an die einzelnen Maschinen erfolgt durch den RuntimeManager in der Regel zufällig. Es gibt jedoch die Möglichkeit, eine eigene Klasse zur Objektverteilung einzusetzen, die z.B. eine lastabhängige Verteilung realisiert.

Wie bereits bei der LU-Dekomposition in Abschnitt 7.1.3 erweist sich die benötigte Kommunikation als ausgesprochen hinderlich für Parallelität, wenngleich die Auswirkungen nicht ganz so drastisch sind, wie Abbildung 7.14 zeigt. Da es keinen dedizierten Verteilerprozeß gibt, muß ein Prozeß sowohl die Verteilung der Daten, als auch die Berechnung seines "Anteils" übernehmen. Dies stellt in der Theorie kein echtes Problem dar, wenn es gelingt, die Verteilungsreihenfolge günstig zu wählen. Da JavaParty nur an solche `remote`-Objekte Kopien der Parameter übergibt, die sich nicht in derselben virtuellen Maschine befinden (vgl. Tabelle 6.2), benötigt die Übergabe von `RT` an den lokalen Worker nur die Dauer einer lokalen Zeigerübergabe (und ist somit vernachlässigbar). Wenn dieser lokale Worker als letzter gestartet wird, beginnt er seine Berechnung beinahe zeitgleich mit dem zuletzt gestarteten entfernten Worker, was vertretbar erscheint.

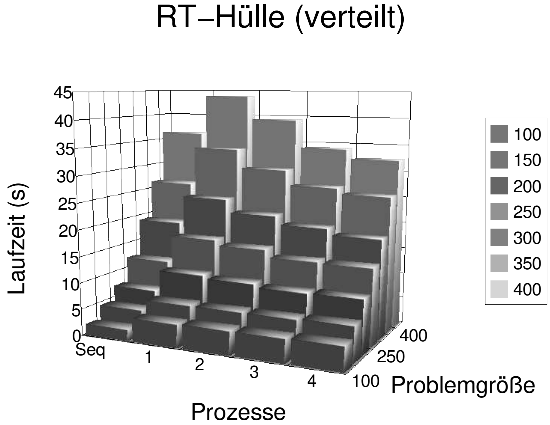

Abbildung 7.14.: Die Laufzeit von RT-Hülle auf dem Testsystem

Die Erzeugung dieser Reihenfolge erweist sich praktisch als durchaus schwer, da die hierfür gedachte Methode `LocalJVM.id` gelegentlich -1 zurückliefert, wenn der Aufruf aus einer Methode eines nicht-`remote`-Objektes erfolgt (was hier allerdings der Fall ist). Dadurch wird der lokale Worker zu früh gestartet und läuft ab dann im Timesharing pseudo-gleichzeitig zur Verteilung der Daten an die übrigen Worker. Es zeigte sich, daß auf diese Art die Verteilung länger dauert, als die lokale Berechnung.

Eine einfache Lösung hierfür ist, zunächst alle Daten zu verteilen und erst anschließend alle Worker zu starten. Dadurch wird zwar (potentielle) Parallelität eingebüßt, weil die zuerst erzeugten Worker unnötig warten müssen, bevor sie die Berechnung beginnen können, praktisch wirkt sich dieser Umstand allerdings bei den betrachteten Problemgrößen nicht negativ aus. Abbildung 7.14 zeigt die gemessenen Laufzeiten.

7.3. Mergesort (dc3)

Mergesort ist ein Sortieralgorithmus nach dem Prinzip des *Divide and Conquer* (DC). Diese Algorithmen lösen ein Problem durch die sukzessive Aufteilung in unabhängige Teilprobleme, wodurch sich interessante Parallelisierungsmöglichkeiten ergeben. Anhand bestimmter Eigenschaften (Verzweigungsgrad, Pfadlänge, etc.), lassen sich DC-Algorithmen in verschiedene Klassen unterteilen, wobei sich für einige Klassen eine allgemeine parallele Lösung angeben läßt [21]. Der hier untersuchte Mergesort-Algorithmus gehört zu einer Klasse, für die eine solche Lösung existiert und erfüllt dabei drei wichtige Einschränkungen:

- Der Kombinationsschritt verzichtet auf die Verwendung der Eingabedaten.
- Alle Pfade des Verzweigungsbaumes sind gleich lang (entspricht gleicher Rekursionstiefe).
- Alle Knoten besitzen den gleichen Verzweigungsgrad.

Aus diesen Einschränkungen ergibt sich das Muster aus Abbildung 7.15 für den allgemeinen parallelen Fall. Zu beachten ist, daß n nicht die Problemgröße (also die Länge der Eingabedaten) sondern die Rekursionstiefe angibt. Die Länge der Eingabearrays ist k^n, wobei für Mergesort der Problemteilungsgrad $k = 2$ gilt. Dadurch ist die Anzahl der Iterationen der äußeren (sequenziellen) Schleifen logarithmisch in der Problemgröße, und somit ist die Anzahl der benötigten globalen Synchronisationen relativ klein.

Dieses Beispiel zeichnet sich durch nicht affine Schleifengrenzen und verschachtelte parallele Schleifen aus. Die Kommunikation erfolgt entlang einer Baumstruktur; am besten eignet sich die Form eines Binomialbaumes, um größtmögliche Lokalität zu erreichen.

Abbildung 7.16 zeigt den Auszug einer sequenziellen Java-Implementierung des Ausgangsprogramms, wobei sich die sequenziellen Schleifen nur durch vorangestellte Kommentare von den parallelen Schleifen der C-Version unterscheiden. Die Portierung von C nach Java war relativ einfach; die einzige Schwierigkeit bestand in der Tatsache, daß in Java kein Referenztyp existiert. Die zusätzliche Indirektionsstufe bei der Übergabe von Parametern (in C über den Operator `&` erreichbar), muß in Java explizit programmiert werden. Dies geschieht entweder durch die Definition einer eigenen Klasse `Reference`, oder pragmatischer durch die Verwendung null-dimensionaler Arrays. So wird aus einer C-Anweisung:

```
basic(&(a2[i]),a1[i]);
```

in Java

```
basic(a2[i],a1[i]);
```

wobei `a2[i]` selbst wieder ein Array geeigneten Typs (z.B. `Object[]` oder `int[]`) ist, und genau ein Element (`a2[i][0]`) besitzt, welches dem Wert von `a2[i]` in C entspricht.

```
input(&(a0[0]));

/* divide phase */
h[0] = a0;
seqfor(m=1;m<=n;m++) {
  parfor(l=0;l<pow(k,m);l++) {
    divide(&(h[m][l]),h[m-1][l/k],l\%k);
  }
}
a1 = h[n]

/* basic phase */
parfor(i=0;i<pow(k,n);i++) {
  basic(&(a2[i]),a1[i]);
}

/* combine phase */
g[0] = a2;
seqfor(m=1;m<=n;m++) {
  parfor(i=0;i<pow(k,n-m);i++) {
    parfor(j=0;i<k;j++) {
      args[m][i][j] = g[m-1][k*i+j];
    }
    combine(&(g[m][i]),args[m][i]);
  }
}
a3=g[n];

output(a3[0]);
```

Abbildung 7.15.: Parallele C-Version des Divide-and-Conquer-Algorithmus

```
...

// combine phase
g[0] = a2;
for (int m=1; m<=n; m++) {
   // parfor:
   for (int i=0; i<(int)Math.pow(k,n-m); i++) {
      // parfor:
      for (int j=0; j<k; j++) {
         args[m][i][j] = g[m-1][k*i+j];
      }
      combine(g[m][i],args[m][i]);
   }
}
a3 = g[n];

...
```

Abbildung 7.16.: Sequenzielle Java-Implementierung des Ausgangsprogramms

Das so portierte Ausgangsprogramm enthält insgesamt vier zu parallelisierende Schleifen: je eine für den `divide`- und `basic`-, sowie zwei verschachtelte Schleifen für den `merge`-Schritt. Der `basic`-Schritt für Mergesort ist die Identität, und diese läßt sich durch eine einfache (sequenzielle) Zuweisung des ganzen Arrays effizienter realisieren als das parallele Kopieren der einzelnen Arrayelemente, zumal die Initialisierung einer parallelen Schleife in der Regel aufwendiger ist als die in Maschinencode realisierte Kopiermethode der Java-Standardbibliothek. Somit bleiben die Schleifen im `divide`- und `merge`-Schritt, von denen letztere besonders interessant sind, da in diesem Schritt die eigentliche Arbeit (also die Sortierung) geleistet wird.

7.3.1. Implementierung mit Titanium

Da Titanium wie bereits erwähnt kein Sprachkonstrukt besitzt, das mit einem `parfor` vergleichbar ist, müssen die parallelen Schleifen geeignet ersetzt werden. Dabei wird abhängig von der eigenen Prozeßnummer und der Anzahl der Prozesse das vom jeweiligen Prozeß zu berechnende Teilstück ermittelt, und dieses in einer sequenziellen `for`-Schleife berechnet:

```
...

 // divide phase
 for (int single m=1; m<=n; m++) {
   // parfor:
   int max = (int)Math.pow(k,m);
```

```
        int tile = max/numProcs;
        int start = tile * myProc;
        int end = (myProc==(numProcs-1))?max:(tile * (myProc+1));

        for (int l=start; l<end; l++) {
            h[m][l] = divide(h[m][l],h[m-1][l/k],l%k);
        }
        Ti.barrier();
        h[m-1] = null;
    }

  ...
```

Durch die globalen Zeiger können die Datenstrukturen der Java-Version unverändert übernommen werden. Wie in anderen Beispielen ist dazu nach dem Programmstart eine Verteilung der Zeiger mittels Broadcast notwendig.

```
    if (myProc == 0) {
      in = new Object[n+1][(int)Math.pow(k,n)];
      in[0][0] = input();
    }

    h = broadcast this.in from 0;
```

Auf eine Verwendung der Titanium-Arraytypen wurde wegen der zu erwartenden schlechteren Laufzeit (näheres in Abschnitt 7.4.1) verzichtet.

Die inneren parallelen Schleifen werden sequenziell behandelt, da Titanium von sich aus nur eindimensionale Prozeßnummern zur Verfügung stellt und sich die Operationen der inneren Schleifen im betrachteten Beispiel auch sequenziell effizient realisieren lassen. Nach jeder parallelen Schleife werden die Prozesse mittels `Ti.barrier` synchronisiert, dazu müssen die Zähler der äußeren Zeitschleife als `single` deklariert werden, was angesichts der vorhersagbaren Schleifengrenzen jedoch kein Problem darstellt.

Die Laufzeit des Programms betrug mit einem Thread auf der Linux Plattform etwa das dreifache der sequenziellen Java-Version. Eine Untersuchung mit mehr als einem Thread scheiterte an einer hartnäckigen und wenig aussagekräftigen Fehlermeldung

```
unimplemented native method: m8getClassmT6Object4lang4java
```

mit der das Programm stets kurz vor dem Ende der Berechnung den Dienst quittierte. Angesichts der geringen Aussagekraft einer konkreten Zeitmessung auf dem zur Verfügung stehenden Einprozessorsystem, wurde auf eine genauere Untersuchung des Fehlers verzichtet. Die rasche Folge, mit der in Berkeley neue Titanium-Versionen veröffentlicht werden, läßt jedoch auf eine baldige Behebung des Fehlers hoffen.

7.3.2. Implementierung mit HighPerformanceJava

Grundlage der Implementierung mit HighPerformanceJava war ebenfalls das nach Java portierte C-Programm aus Abbildung 7.15. Da HighPerformanceJava keine verschachtelten parallelen Schleifen unterstützt, bleiben von den ursprünglich vier Schleifen des Ausgangsprogramms letztendlich zwei zu parallelisierende Schleifen übrig: die Schleife des `divide`-Schritts und die äußere Schleife des `merge`-Schritts. Diese werden im Quelltext einfach mit entsprechenden Kommentaren versehen:

```
// combine phase
g[0] = a2;
for (int m=1; m<=n; m++) {
    int high = (int)Math.pow(k,n-m);

    /*par threads=1*/  // bzw.: /*par threads=128*/
    for (int i=0; i<high; i++) {

        for (int j=0; j<k; j++) {

  ...
```

Da HighPerformanceJava keine Methodenaufrufe innerhalb des Rumpfes paralleler Schleifen erlaubt, war es (wie zuvor im Beispiel 7.2.2) nötig, die enthaltenen Methodenaufrufe manuell durch den aufgerufenen Code zu ersetzen. Durch diese Veränderung können alle benutzten Arrays um eine Dimension reduziert werden, da diese nur benötigt wurde, um die Parameterübergabe per Referenz aus dem zugrundeliegenden C-Programm zu simulieren. Da jede Arraydimension in Java mit der Erzeugung zusätzlicher Objekte für die Verwaltung verbunden ist, reduziert sich der Speicherplatzbedarf des Programms auf weniger als die Hälfte der Ausgangsversion.

Der Precompiler `javar` generiert aus dem so modifizierten Programm das (Thread-) parallele Zielprogramm, das anschließend mit einem beliebigen Java-Compiler übersetzt werden kann. Abbildung 7.17 zeigt die Laufzeit des Programms auf der Linux-Plattform. Die Spalten "HPJ-1" und "HPJ-128" zeigen die Ausführung mit einem, bzw. 128 Threads, die allerdings im Timesharing, und somit nicht wirklich parallel laufen. Zum Vergleich zeigt die erste Spalte die Laufzeit des sequenziellen Java-Programms. Da die Parallelisierungsanweisungen für `javar` in Form von Kommentaren im Quelltext angegeben werden, sind die Quellprogramme dieser Meßreihen textuell identisch.

Umso mehr überascht die Verringerung der Laufzeit der HighPerformanceJava-Version mit einem Thread im Vergleich zur sequenziellen Java-Version. Zwar werden im generierten Zielprogramm nur $2 \log N$ Threads erzeugt, dies ist aber dennoch ein echter Mehraufwand gegenüber der sequenziellen Version. Eine mögliche Erklärung für dieses Verhalten liegt in der Art der Speicherverwaltung in Java und der Arbeitsweise der JIT-Compiler. Durch das "Inlining" der Methodenaufrufe entsteht eine Methode mit relativ hohem Speicherbedarf. Da Java den Laufzeitstack in sogenannte *Frames* unterteilt

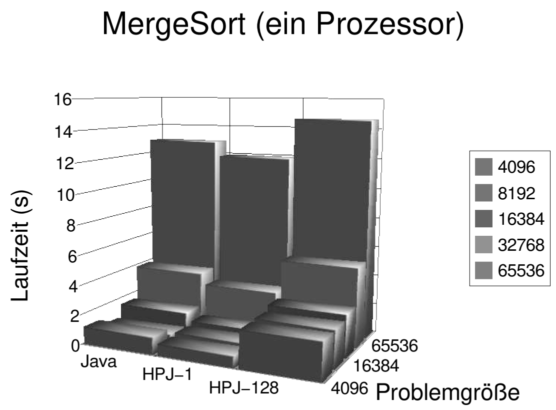

Abbildung 7.17.: Die Laufzeit von Mergesort mit HighPerformanceJava

(jeder dieser Rahmen entspricht dabei in etwa einem Methodenaufruf), wird dieser Speicher in der Java-Version des Programms erst nach Beendigung der gesamten Abarbeitung wieder freigegeben.

Im Gegensatz dazu lagert `javar` den Code im Rumpf der parallelisierten Schleifen in einzelne Methoden und diese wiederum in eigene Klassen aus. Dieses Vorgehen begünstigt zum einen die automatische Speicherbereinigung, zum anderen arbeiten JIT-Compiler auf Klassenebene und kompilieren dabei bevorzugt solche Methoden, die besonders häufig aufgerufen werden. Der Hotspot JIT-Compiler von SUN z.B. kompiliert Methoden erst, wenn diese wenigstens zum zweiten mal durchlaufen werden. Dies würde in diesem Beispiel dazu führen, daß die Java-Version ganz ohne JIT-Kompilation auskommen müßte. Der in diesem Beispiel benutzte JIT-Compiler zeigt den Messungen zufolge ein ähnliches Verhalten.

Das Ergebnis liefert somit zusätzlich die Aussage, daß die Laufzeit von Java-Programmen stark vom benötigten Speicherplatz und dessen Aufteilung abhängt. Vergleichsweise gering fällt dagegen der Overhead aus, der für die Thread-Erzeugung und -verwaltung benötigt wird, wie die dritte Spalte des Diagrams zeigt.

7.3.3. Implementierung mit JavaParty

JavaParty ermöglicht wegen des dynamischen Threadmodells eine Reihe unterschiedlicher Implementierungen. Zum einen läßt sich, wie bereits in Abschnitt 7.1.3 beschrieben, eine SPMD-artige Parallelität erzeugen; zum anderen können die parallelen Schleifen dynamisch verteilt werden, was der Methode aus dem letzten Abschnitt entspricht. Allerdings sind hierbei auch verschachtelte Parallelschleifen möglich.

Der Abschnitt 7.3.3 beschreibt eine Version, die sich mit sehr geringen Veränderungen des Ausgangsprogramms realisieren läßt; zusätzlich wird in Abschnitt 7.3.3 eine rein objektorientierte, rekursive Version präsentiert, die den besonderen Eigenschaften von

JavaParty besser entspricht.

Iterative Version

Mit JavaParty ist es möglich, die Titanium-Version aus Abschnitt 7.3.1 beinahe unverändert zu übernehmen. Der Broadcast der Zeiger zum Programmstart wird mit der in Abschnitt 7.1.3 vorgestellten Technik durch eine globale Datenstruktur ersetzt. Zudem wird durch ein einmaliges Verteilen des Programms auf alle verfügbaren Maschinen eine SPMD-artige Parallelität erzeugt.

Die globale Synchronisation läßt sich auch hier mit der ebenfalls in Abschnitt 7.1.3 vorgestellten `barrier`-Implementierung erreichen. Insgesamt liegt die Laufzeit jedoch auch hier weit hinter der des sequenziellen Programms, was auf die hohen Kommunikationskosten zurückzuführen ist.

Rekursive Version

Aus diesem Grund wurde eine weitere (rekursive) Version des Divide-And-Conquer-Algorithmus realisiert, die stärker objektorientiert ist, und somit der Ausrichtung von JavaParty besser entspricht.

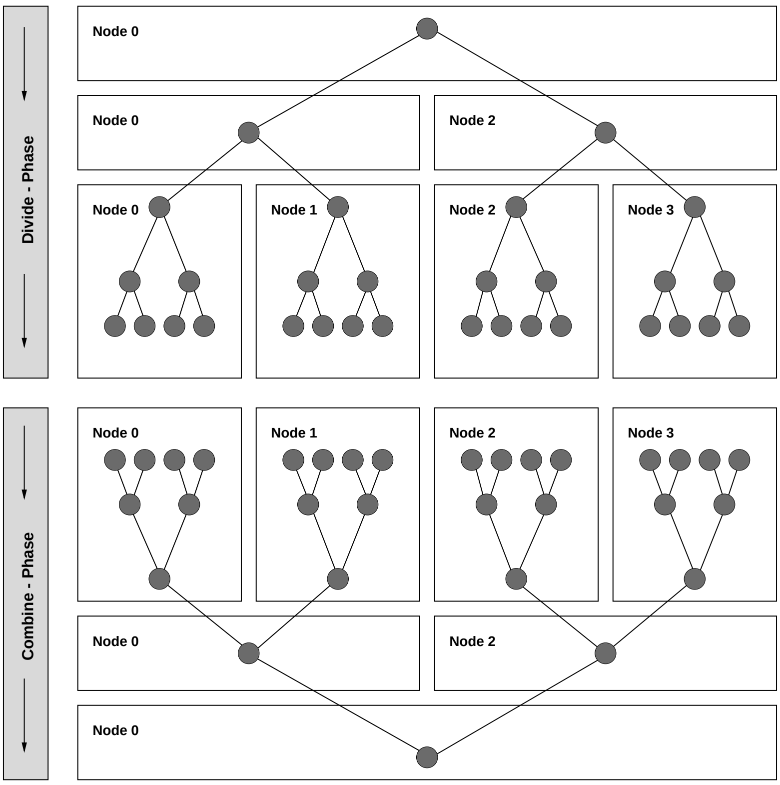

Abbildung 7.18.: Verteilung des Algorithmus auf vier Prozessoren

Der Algorithmus benutzt eine als `remote` deklarierte Klasse `MergeSort`, um ein Array $[a_0 \ldots a_{n-1}]$ mit n Elementen rekursiv zu sortieren. Die Verarbeitung durchläuft, wie bei Divide-And-Conquer üblich, die Schritte `distribute`, `basic` und `combine`, wobei die eigentliche Sortierung (in diesem Fall, das sortierte Zusammenführen von zwei vorsortierten Arrays) in der Funktion `combine` erfolgt. Der wesentliche Unterschied zur iterativen Version besteht in der Funktion `divide`. Hier wird zunächst das Eingabearray in zwei Hälften zerlegt; anschließend wird für jede der beiden Hälften ein neues Objekt `MergeSort` instantiiert. Die Rekursion endet, sobald die Listen einelementig (und somit trivialerweise sortiert) sind. Abbildung 7.18 zeigt schematisch die Arbeitsweise des Algorithmus.

```
int length=size/base;
int me = jp.lang.DistributedRuntime.thisLocation();
int nodes = jp.lang.DistributedRuntime.machines();

for (int i=0; i<base; i++) {
    int[] out = new int[length];
    int stride = (int)(nodes/Math.pow(base,depth+1));

    System.arraycopy(data,i*length,out,0,length);

    jp.lang.DistributedRuntime.setTarget( (me+(i*stride)) % nodes);
    children[i] = new MergeSort(base,out,depth+1);
}
```

Abbildung 7.19.: Die veränderte Methode `distribute`

Die hier vorgestellte Realisierung macht sich zwei Stärken von JavaParty zunutze:

- Zum einen ist JavaParty (anders als RMI) in der Lage, Methoden eines `remote`-Objekts ähnlich effizient aufzurufen, wie die eines lokalen Objektes, falls sich das Objekt innerhalb der lokalen virtuellen Maschine befindet.

- Zum anderen erweist sich die Semantik der Parameterübergabe als überaus handlich im Gebrauch: während `remote`-Objekte, in unserem Fall also untergeordnete `MergeSort`-Objekte per Referenz übergeben werden, werden alle anderen Objekte (also z.B. die Subarrays, die die Eingabedaten beinhalten) an `remote` Objekte per Wert übergeben. Dieses zunächst etwas verwirrende Verhalten (die Details finden sich in Abschnitt 6.4) liefert hier jedoch exakt das gewünschte Ergebnis: die Eingabedaten werden als Block kopiert (schnellere Übertragung), die Referenzen auf entfernte Objekte verhalten sich jedoch wie "echte" Referenzen (wichtig für die richtige Synchronisation[10]).

[10] Die Synchronisation auf `remote` Objekte ist dennoch nicht ganz unproblematisch, vgl. Abschnitt 6.3

Wie im Mittelteil von Abbildung 7.18 zu sehen ist, läßt sich die Objektverteilung optimieren, indem nur Objekte auf anderen Maschinen erzeugt werden, solange die Breite des Baumes kleiner ist, als die Anzahl der verfügbaren Maschinen. Die Implementierung der Methode `distribute` zeigt Abbildung 7.19.

Zu beachten ist, daß JavaParty jedes neue `remote`-Objekt (Konstruktoraufruf in der letzten Zeile) auf einer beliebigen Maschine plaziert. In diesem Beispiel wird deshalb mittels `DistributedRuntime.setTarget` explizit eine Zielmaschine vorselektiert, um die Verteilung aus Abbildung 7.18 sicherzustellen.

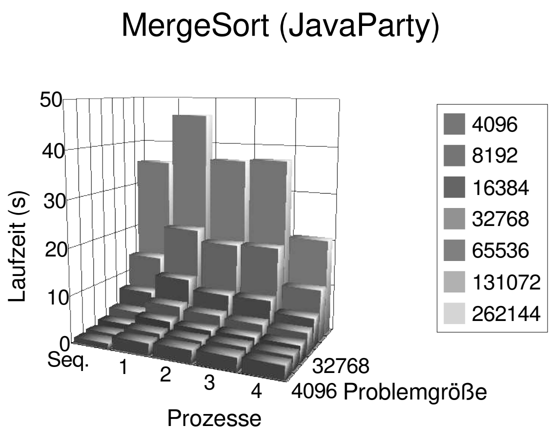

Abbildung 7.20.: Die Laufzeit von Mergesort auf dem Testsystem

Abbildung 7.20 zeigt die gemessenen Laufzeiten auf der Testplattform. Hierbei läßt sich deutlich erkennen, daß der Einsatz von 3 Maschinen keinen Vorteil gegenüber dem Einsatz von 2 Maschinen bewirkt. Dies ist auf die Struktur des Algorithmus zurückzuführen. Der Rekursionsbaum besitzt auf jeder Ebene h stets 2^h Knoten. Ist N die Größe des Eingabearrays, so werden für eine sinnvolle Verteilung $M' = 2^{h'}$, $(0 \leq h' \leq \log N)$ Prozessoren benötigt.

Falls die Anzahl M der vorhandenen Prozessoren keine Zweierpotenz ist, also $2^{h''} < M < 2^{h''+1}$ $(0 \leq h'' < \log N)$, so bleiben entweder einige Maschinen ungenutzt (falls $M > M'$), oder aber es gibt wenigstens eine Maschine, die mehr als ein Teilproblem zu lösen hat (falls $M < M'$). Werden die Kommunikationskosten vernachlässigt, ergibt sich im ersten Fall die gleiche Laufzeit wie mit M' $(M' < M < 2^{h''+1})$ Maschinen; im zweiten Fall entspricht die Laufzeit der von $2^{h''}$ $(2^{h''} < M < M')$ Maschinen.

7.4. Laufzeitvergleich im Sequenziellen

Aus technischen Gründen war es auf der benutzten Testplattform leider nicht möglich, mit Titanium und HighPerformanceJava echte Beschleunigung zu erreichen. Titanium erzeugt nur mit dem Split-C Backend Programme, die sich verteilt ausführen lassen. Die zugrundeliegende Split-C Bibliothek war für die vorhandene Hardware jedoch nicht verfügbar. Die mit den vorhandenen Backends erzeugbaren Programme erreichen, ebenso wie die mit HighPerformanceJava erstellten Programme, nur auf SMP-Architekturen echte Beschleunigung. Da die zur Verfügung stehenden SMP-Rechner jedoch mit maximal zwei Prozessoren ausgestattet sind, und zudem als Server genutzt werden (somit also ein relativ ungleichmäßiges Lastprofil aufweisen), wurde auf eine Messung verzichtet.

Um dennoch einen ungefähren Vergleich zu ermöglichen, wurden einige Programme mit den zu untersuchenden Sprachen im Vergleich zu der entsprechenden Version in C/C++ oder Java untersucht. Welchen Algorithmus das betrachtete Programm implementiert ist, da hier nur die sequenzielle Version zum Einsatz kommt, von geringer Bedeutung. Dennoch sollte es sich um Schleifenprogramme handeln, die vorwiegend auf Arrays als Datenstruktur arbeiten. Da dies für alle bereits untersuchten Algorithmen der Fall ist, erfolgte die Auswahl beliebig unter den bereits vorhandenen Beispielen.

7.4.1. Titanium

Titanium übersetzt (wie bereits erwähnt) seine Quellprogramme zunächst in C und im weiteren in Maschinencode. Durch die Bereitstellung der wichtigsten Java-Bibliotheken sollte sich Titanium für einfache Java-Programme somit auch als reiner Java-Compiler verwenden lassen. Die Hoffnung war, auf diese Art eine Laufzeit zu erreichen, die deutlich geringer ist, als die der interpretierten Bytecodeversion.

Da für alle bereits untersuchten Algorithmen bereits eine sequenzielle Java-Implementierung existierte, wurde zunächst(o.B.d.A) das Java-Programm der LU-Dekomposition unverändert mit Titanium übersetzt. Erstaunlicherweise lag die Performance auf dem Linux-System etwas hinter der reinen Java-Version[11]. Erst mit deaktiviertem JIT-Compiler war die Bytecodeversion langsamer, als das mit Titanium übersetzte Programm (vgl. Abbildung 7.21). Der Vergleich ist allerdings nicht ganz fair, da es sich bei der Java-Version um ein rein sequenzielles Programm handelt. Obwohl die Quellprogramme textuell identisch sind, erzeugt der Titanium-Compiler daraus ein (potentiell) verteilbares Programm mit dem entsprechenden Overhead.

Der Grund für dieses Ergebnis liegt also vermutlich zum Teil in dem Verwaltungsaufwand für die globalen Referenzen. Möglicherweise findet auch keine automatische Optimierung bei Zugriffen auf lokale Objekte statt. Zudem sind die aktuellen JIT-Compiler relativ gut entwickelte, hochspezialisierte Programme, die auf die besonderen Anforderungen von reinem Java wesentlich besser optimiert wurden, als man dies von einem prototypischen Compiler für eine von Java abgeleitete Sprache erwarten darf.

Ein weiterer Erklärungsversuch kann indes ausgeschlossen werden: da die Implementierung zweidimensionale (dynamische) Java-Arrays sowohl für die Ein- als auch Ausgabedaten benutzt, stellte sich die Frage, ob die zusätzliche Indirektionsstufe bei jedem Zugriff möglicherweise für die Leistungseinbußen verantwortlich ist. Da Titanium einen eigenen, echt mehrdimensionalen Array-Typ zur Verfügung stellt, wird der Code für "gewöhnliche" Java-Arrays vermutlich von Titanium weniger stark optimiert, als

[11] Für kleine Problemgrößen ist die Laufzeit etwas besser, da für jedes Java-Programm zunächst die Laufzeitumgebung (VM) gestartet werden muß. Dieser Mehraufwand ist unabhängig von der Problemgröße.

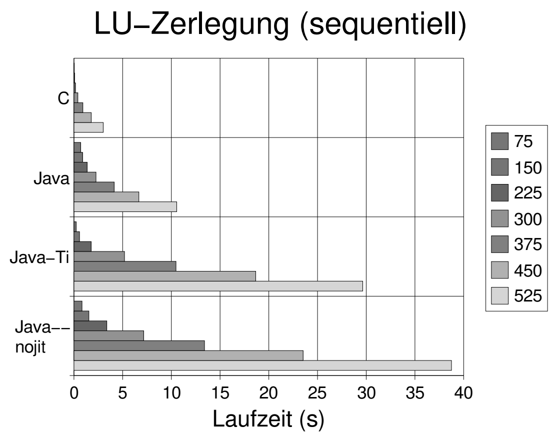

Abbildung 7.21.: Laufzeiten der LU-Zerlegung im Vergleich

JIT-Compiler dies tun.

Versuchsweise wurde der Algorithmus nun unter Verwendung von Titanium-Arrays und Domänen realisiert. Dabei zeigte sich, daß die vielen für diese Arrays definierten Operationen (`fill`, `split`, etc.) durchaus nützlich sind. Der Iterator `forall`, der alle Elemente einer Domäne besucht, ist prädestiniert für die Initialisierung mehrdimensionaler Arrays. Anders als erwartet, brachte jedoch auch die Verwendung von Titanium-Arrays sogar Geschwindigkeitseinbußen, wie Abbildung 7.22 zeigt.

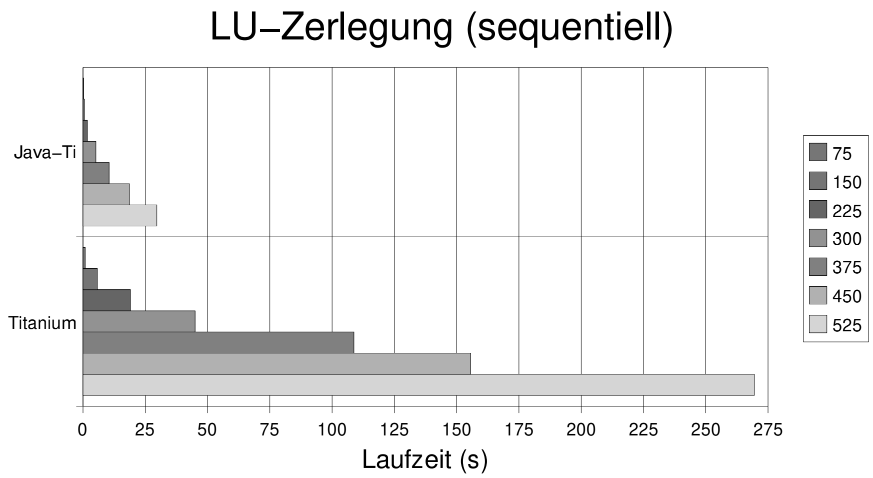

Abbildung 7.22.: Laufzeiten der LU-Zerlegung im Vergleich

7.4.2. Java

Die Messungen aus Abschnitt 7.4.1 ergaben, daß Java mit aktiviertem JIT-Compiler in dem betrachteten Beispiel um etwas mehr als den Faktor 3 langsamer ist als ein reines C/C++ Programm. Dies ist allerdings ein durchaus akzeptabler Wert, wenn man die Unterschiede der Arrayimplementierung betrachtet: mehrdimensionale Arrays lassen sich in C als zusammenhängender Speicherbereich belegen, wobei sich mehrere Arraydimensionen durch explizite Adreßberechnung bei jedem Zugriff auf eine reduzieren lassen; eine Überprüfung der Indexgrenzen findet nicht statt. In Java werden mehrdimensionale Arrays zum einen als Array von Arrays realisiert, was eine zusätzliche Indirektionsstufe für jede Dimension bewirkt, zum anderen wird vor jedem Zugriff die Einhaltung der Indexgrenzen überprüft. Beides zusammen rechtfertigt (wenigstens zum Teil) die geringere Effizienz des Java-Programms.

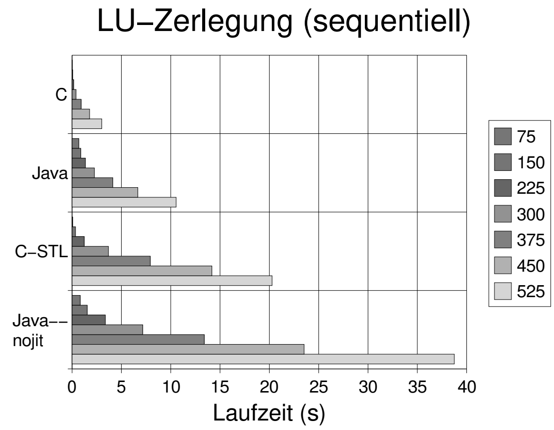

Abbildung 7.23.: Laufzeiten der LU-Zerlegung im Vergleich

Abbildung 7.23 zeigt die Java-Version im Vergleich zu einer C-Version mit dynamischen Arrays. Die STL (C++ Standard Template Library) stellt einen entsprechenden Typ `Vector` bereit, der ebenfalls eine Überprüfung der Indexgrenzen erlaubt. Wie sich zeigt, ist diese Version ihrerseits um etwa den Faktor 2 langsamer als die Java-Version, deren Arrays allerdings auch nicht dynamisch sind. Insgesamt liegt die Leistung des Java-Programms also durchaus in einem vernünftigen Bereich.

7.4.3. LooPo

Obwohl Schleifenprogramme mit Hilfe von JIT-Compilern im allgemeinen gute Laufzeiten erzielen, scheint dies nicht in jedem Fall für Schleifenprogramme zu gelten, die von LooPo generiert wurden. Wie Abbildung 7.24 zeigt, liegt die Laufzeit des von LooPo erzeugten Programms hinter der des reinen Java-Programms wenn beide sequenziell auf einem Rechner ausgeführt werden.

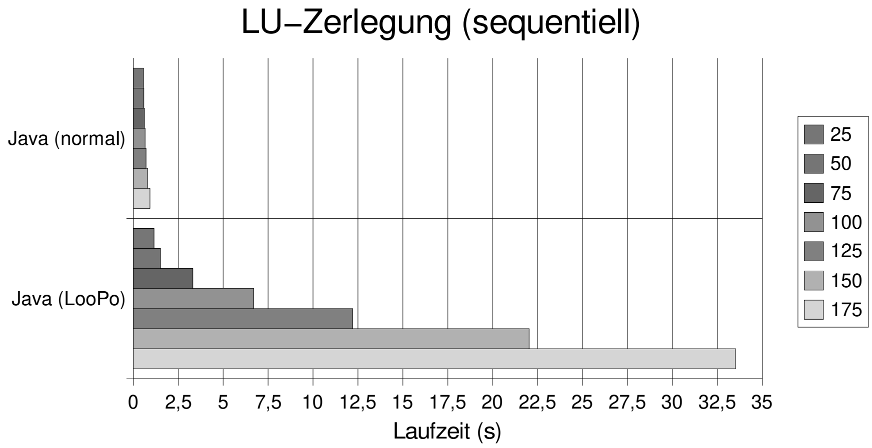

Abbildung 7.24.: Laufzeiten der LU-Zerlegung im Vergleich

Der Grund ist unter Umständen die reichlich komplexe Struktur der LooPo-Programme, die eine große Zahl von Fallunterscheidungen, Typkonvertierungen und Methodenaufrufen enthalten, die im ursprünglichen Programm nicht vorkommen. Eine Verschlechterung der Laufzeit aufgrund von Fallunterscheidungen liegt (zumindest bei C/C++-Programmen) zumeist an Cache-Effekten; ob dies für Java-Programme übertragbar ist, ist jedoch fraglich. Es ist kaum anzunehmen, daß eine Java-VM die gleichen Optimierungen (Caching, Sprungvorhersage) softwareseitig nachbildet, die moderne CPUs als Hardware realisieren.

Viel mehr ins Gewicht fallen da wohl eher die Typkonvertierungen (die in Java stets zur Laufzeit erfolgen) und die mathematischen Funktionsaufrufe (`ceil`, `floor`, etc.). Zwar besitzen die meisten Java-Compiler eine Optimierungsfunktion, die solche Methoden "inline" ersetzt. Dazu müssen diese aber als `private` oder `final` deklariert sein. Zumindest ersteres ist natürlich für Bibliotheksfunktionen nicht erfüllbar.

Dennoch scheint es möglich zu sein, einen Teil dieses Mehraufwands zu optimieren, wie Abbildung 7.25 zeigt. Die erste Zeile enthält die Laufzeit des LooPo-Programms für Titanium; die zweite zeigt die Laufzeit des LooPo Java-Programms, welches auch in der letzten Zeile von Abbildung 7.24 dargestellt ist. Die Verschlechterung gegenüber dem Titanium-Programm ist aufgrund der geringen Problemgröße dem Aufwand zum Starten der Java-VM zuzuschreiben. Die letzte Zeile zeigt schließlich die Laufzeit des LooPo-Programms für JavaParty.

Festzuhalten ist, daß Titanium scheinbar in der Lage ist, LooPo-Programme so weit zu optimieren, daß sogar die in Abschnitt 7.4.1 untersuchte Verwendung der Titanium-Arrays keine effektive Verschlechterung bewirkt. Eine deutliche Verbesserung der Laufzeit sollte sich zudem durch eine geeignetere Anpassung der Codegenerierung an Java erreichen lassen. Vorschläge wären eine Reduzierung der Typkonvertierungen und ein explizites "Inlining" der mathematischen Funktionen.

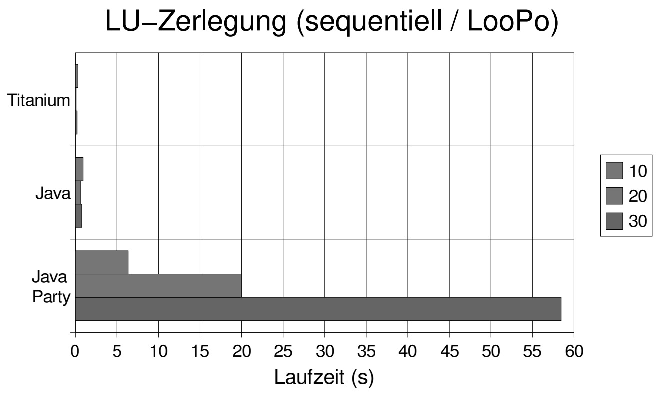

Abbildung 7.25.: Laufzeiten der LU-Zerlegung im Vergleich

7.5. Zusammenfassung

Wie die durchgeführten Tests zeigen wird die Laufzeit von parallelen Java-Programmen stark von zahlreichen Randeffekten beeinflußt. Vor allem der Speicherbedarf und die Art der Kommunikation sind entscheidende Faktoren. Tabelle 7.2 zeigt eine Übersicht über die implementierten Algorithmen.

	LU-Zerlegung	**RT-Hülle**	**Mergesort**
Art	Numerischer Algorithmus	Graphalgorithmus	Sortieralgorithmus
Schleifen im Quellprogramm	Fünf Schleifen, nicht perfekt verschachtelt, affine Grenzen.	Drei Schleifen, perfekt verschachtelt, dynamische Grenzen.	Vier Schleifen, teilweise perfekt verschachtelt, vorhersagbare Grenzen
Parallelisierung	LooPo	Task-Farming [19]	Allgemeine parallele Version [21]

Tabelle 7.2.: Beispielalgorithmen

Nicht alle Beispiele liesen sich auf Anhieb mit allen Projekten untersuchen. Zum Teil waren Ergänzungen der Funktionalität erforderlich; in zwei Fällen gelang eine Umsetzung nicht. Tabelle 7.3 zeigt die in den einzelnen Abschnitten untersuchten Algorithmen.

Die Ausrichtung und Funktionalität der untersuchten Sprachen unterscheidet sich teilweise sehr stark, so daß ein direkter Vergleich nur schwer möglich ist. Tabelle 7.4 zeigt dennoch eine vereinheitlichte Darstellung der gewonnenen Erkenntnisse.

Titanium erwies sich in der Handhabung als relativ komfortabel. Wünschenswert wäre ein verteilter Arraytyp, oder wenigstens eine einfache Möglichkeit, Daten explizit zu verteilen. Während Titanium als reiner Java-Compiler keinen Geschwindigkeitsvor-

		Titanium	HighPerformanceJava	JavaParty
LU	**LooPo**	7.1.1	7.1.2	7.1.3
	- Anmerkungen	keine verteilten Arrays	asynchrone Version nicht realisierbar	kein parfor, kein Barrier, keine verteilten Arrays
RT	**Task-Farming**	7.2.1	7.2.2	7.2.3
	- Anmerkungen	kein Iterationspool	nicht möglich	kein parfor, kein Iterationspool
MS	**it3**	7.3.1	7.3.2	7.3.3
	- Anmerkungen	kein parfor		kein parfor
	rekursiv			7.3.3
LU ≡ LU-zerlegung, RT ≡ RT-Hülle, MS ≡ Mergesort				

Tabelle 7.3.: Implementierung der Beispielalgorithmen

teil gegenüber modernen JIT-Compilern bewirkt, kann die statische Analyse und Optimierung von mit LooPo erzeugten Programmen die Laufzeit unter Umständen positiv beeinflussen.

HighPerformanceJava beinhaltet einige interessante Ideen (Iterationspool, globale Synchronisationsvektoren) jedoch lassen sich diese aufgrund zahlreicher Einschränkungen nicht immer optimal nutzen. In zwei Beispielen gelang eine Umsetzung mit HighPerformanceJava nicht. Da sich HighPerformanceJava nicht verteilt nutzen läßt, sind die Einsatzmöglichkeiten für die hier untersuchten Fälle eher gering.

HPJava bietet für nahezu alle beobachteten Probleme eine komfortable Lösung. Es verfügt über parallele Sprachkonstrukte ebenso, wie über globale und verteilte Datentypen. Es läßt sich verteilt auf Standard-VMs ausführen und bietet uneingeschränkten Zugriff auf alle Java-Funktionen sowie die Funktionalität der zugrundeliegenden Kommunikationsbibliothek. Diese ist auf der Basis von Standard-MPI Bibliotheken realisiert und verspricht einfache und effiziente Kommunikation. Umso mehr ist es schade, daß es bis auf die Schnittstelle zu MPI noch keine Implementierung dieser Konzepte gibt.

JavaParty verfolgt einen stark objektorientierten Ansatz und läßt sich (entsprechende Javakenntnisse vorausgesetzt) sehr bequem für die manuelle Erstellung paralleler Programme nutzen. Die Kosten der impliziten Kommunikation erwiesen sich als zu hoch für den generellen Einsatz mit automatisch generierten Programmen. LooPo Programme mit expliziter und (im Idealfall) blockweiser Kommunikation könnten dieses Problem lösen. Da JavaParty ausschließlich auf Java basiert ist, war es die einzige der untersuchten Sprachen, mit der verteilte Programme auf der Testplattform untersucht werden konnten.

	Java	**Titanium**	**HighPerformance-Java**	**HPJava**	**JavaParty**
Zielsprache	Bytecode	Maschinencode	Bytecode	Bytecode + Bibliotheken	Bytecode
Plattformen					
- gemeinsamer Speicher	Java VM	Uniproc-, pthread- oder SMP-Backend	Java VM	Java VM	Java VM
- verteilt	nein	Split-C Backend	nein	mehrere Java VMs + AdLib + MPI	mehrere Java VMs + RMI
- lokal verfügbar	Solaris jdk1.2.2	Uniproc- und pthread-Backend	ja	nur MPIJava	ja
Parallelität					
- zentrales Konzept		SPMD	Taskfarming	Datenparallelität	RMI
- Erzeugung durch		Prozesse	Threads	Prozesse (VMs)	Prozesse (VMs)
- Prozessmodell		statisch	dynamisch	statisch	statisch
- Dynamische Threaderzeugung möglich	ja	nein	ja	ja	ja
- Einschränkungen			keine verschachtelten Schleifen, keine dynamischen Schleifengrenzen, keine Methodenaufrufe aus parallelen Schleifen	kein wahlfreier Zugriff auf verteilte Arrays	kein parfor, kein Barrier
Gemeinsamer Speicher					
- zentrales Konzept		DSM-Simulation auf Zeigerebene	physikalisch gemeinsamer Speicher (SMP)	Globale und Verteilte Arrays, globale Skalare	Globale Objekte
- standard Sichtbarkeit		global	global (innerhalb einer VM)	lokal; Ausnahme: als global bzw. verteilt deklarierte Variablen	lokal; Ausnahme: als `remote` deklarierte Objekte

	Java	**Titanium**	**HighPerformance-Java**	**HPJava**	**JavaParty**
- Einschränkungen		keine verteilten Datentypen			keine globalen/verteilten Arrays
Einsatz-möglichkeiten					
- manuelle Parallelisierung		ja	nicht in allen Fällen erfolgreich		ja
- Zielprache für automatische Codegenerierung		einfach	teilweise aufwendige Nachbearbeitung		einfach

Tabelle 7.4.: Ergebnisse aus den Fallstudien

8. Andere Projekte und Literaturhinweise

Am Ende dieser Arbeit sei noch kurz auf einige Projekte hingewiesen, die in diesem Zusammenhang von Interesse sein können. Da die Zahl der Arbeiten zu diesem Thema unüberschaubar ist, repräsentiert diese Liste nur eine willkürliche, und in keiner Weise vollständige Auswahl.

8.1. Enterprise Java Beans

Diese von SUN vorgestellte Spezifikation[1] soll die Entwicklung Plattformunabhängiger Java-Komponenten (Enterprise Java Beans, kurz EJB) ermöglichen, die sich in verteilten Laufzeitumgebungen (Anwendungsserver oder engl. Application Server) unterschiedlicher Hersteller ausführen lassen. Diese Anwendungsserver stellen dem Entwickler bestimmte Basisdienste zur Verfügung (Kommunikation, zentrale Resourcenverwaltung,...) und sind in der Lage, EJB transparent auf mehreren Rechnern auszuführen Die Kommunikation erfolgt dabei CORBA-basiert.

Ähnlich wie bei JavaParty (vgl. Kapitel 6) werden die kommunikationsspezifischen Codeteile vom Anwendungsserver automatisch generiert. In einem *Deployment* genannten Vorgang werden die EJB als Bytecode auf den Anwendungsserver übertragen und in neu erzeugte Objekte "verpackt". Dies ist erforderlich, da die EJB üblicherweise in kompilierter Form als Bytecode vorliegen, der Quelltext für eine Analyse also nicht mehr zur Verfügung steht. Ein Aufruf von außen wird stets zuerst an das umgebende Server-Objekt und von diesem an die enthaltene EJB weitergeleitet. An dieser Stelle bieten sich natürlich genau die Optimierungen an (Lokalität, Zugriff auf Objektvariablen, etc.), die in Kapitel 6 präsentiert wurden.

Ein potentieller Vorteil dieser Technik ist, daß nur die Beans selbst im plattformunabhängigen Bytecode vorliegen müssen. Die Serverimplementierung, inklusive der CORBA-Schicht, können hingegen in Maschinencode und somit entsprechend effizient realisiert werden. Auch bietet der Deployment-Vorgang Raum für zusätzliche Optimierungen.

8.2. Pangaea

An der Freien Universität Berlin wird unter dem Namen Pangaea ein System zur automatischen Verteilung von Java-Programmen entwickelt. Eine statische Quelltextanalyse soll

[1] `http://java.sun.com/products/ejb/`

dabei Optimierungen erlauben, die für eine rein laufzeit-basierte Automatik unmöglich sind. Der Aspekt der Verteilung soll für den Programmierer transparent sein, das bedeutet: sowohl die Entscheidung für eine bestimmte Verteilungsstrategie, als auch deren programmtechnische Umsetzung geschieht in Pangaea automatisch.

Als mögliche Zielplattform wird neben RMI und CORBA ausdrücklich JavaParty (vgl. Abschnitt 6) hervorgehoben. Durch den modularen Aufbau ist Pangaea jedoch auch für andere Verteilungsplattformen adaptierbar [34].

8.3. FarGo

FarGo untersucht vor allem die Möglichkeiten von Objektmigration zur Lastverteilung in weitverteilten parallelen Anwendungen. Ziel ist es, Programmkomponenten, die mehrere laufende Threads beinhalten können, dynamisch zu verschieben, ohne daß die Verwendung von Ressourcen (Filehandles, etc.) eingeschränkt werden muß.

Eine lauffähige Version, sowie umfangreiche Dokumentation können über die Webseite `http://www.dsg.technion.ac.il/fargo/` bezogen werden. Die verfügbare Version setzt eine Java-VM Version 1.1 voraus; an einer Portierung auf Version 1.2 wird gearbeitet. Schwierig dürfte dabei sein, daß einige Methoden, die zum Anhalten und neu Starten von Threads benötigt werden, in der Version 1.2 nicht mehr vorgesehen sind [22].

8.4. JavaGrande

JavaGrande[2] (früher ACM Workshop on Java for High-Performance Network Computing) ist eine Konferenzreihe, die die Analyse und Erweiterung von Java im Hinblick auf das wissenschaftliche Hochleistungsrechnen zum Thema hat. In den Veröffentlichungen finden sich neben in dieser Arbeit vorgestellten Projekten eine ganze Reihe ähnlicher und/oder weiterführender Themen. Eine (ebenfalls willkürliche und in keiner Weise vollständige) Auswahl sei hier kurz erwähnt:

- Aufgrund durchgeführter Messungen mit Java-Programmen mit und ohne JNI-Unterstützung (Java Native Interface) kommt [35] zu dem Ergebnis, daß reine Java-Bibliotheken in Spezialfällen um den Faktor 15 bis 115 langsamer sind als reine Maschinencode-Implementierungen.

- Eine Java-Umgebung zur Entwicklung verteilter Programme wird in [13] vorgestellt. Über eine Art "Proxy-Architektur" soll der transparente Zugriff auf entfernte Java-Objekte ermöglicht werden. Dies entspricht im Kerngedanken dem, was in JavaParty realisiert ist.

- [24] enthält die Beschreibung eines Systems namens "Charlotte", welches auf der Grundlage von Java eine (virtuelle) DSM Architektur zur Verfügung stellt; durch Fehlertoleranz soll das System auch für Internet-Computing einsetzbar sein.

[2] `http://www.javagrande.org/`

- [33] beschreibt die Probleme bei der Optimierung von Java-Programmen, die aufgrund des Java-Speichermodells entstehen, und schlägt entsprechende Lösungen vor.

9. Abschließende Bemerkungen

Java hat sicherlich auch im Bereich der Schleifenparallelisierung seine Anwendungsmöglichkeiten. Unter entsprechenden Voraussetzungen sind die erzielbaren Ergebnisse zumindest qualitativ durchaus mit denen von C/C++ Programmen vergleichbar. Im weiteren werden die Vor- und Nachteile von Java speziell für den hier untersuchten Bereich kurz zusammengefaßt; anschließend werden einige konkrete Vorschläge zur Erweiterung von LooPo präsentiert, die die automatische Erzeugung paralleler Java-Programme vereinfachen sollen. Am Ende dieser Arbeit finden sich noch einige Bemerkungen zu aktuellen Entwicklungen.

9.1. Hochleistungsrechnen mit Java

Java eignet sich durch seine Plattformunabhängigkeit für den Einsatz auf heterogenen (und weiträumigen) Netzwerken und ist somit eine interessante Zielsprache für massivparallele Anwendungen. Die Java-Version von "Seti-At-Home"[1] ist ein Beispiel für die Verteilung von Berechnungen auf hunderttausende javafähiger Computer. Durch die in Java integrierten Sicherheitskonzepte ist der "Wirtsrechner" dabei gut vor dem "Gastprogramm" geschützt. Dieser Umstand steigert die allgemeine Bereitschaft, nicht benötigte Rechenleistung anderen Anwendern zur Verfügung zu stellen. Java-Programme lassen sich zentral übersetzen und verwalten; die Bindung erfolgt zur Laufzeit, wenn nötig über eine HTTP-Verbindung. Dadurch wird die Entwicklung und Administration solcher Anwendungen stark vereinfacht.

Nachteilig für die Laufzeit von Java-Programmen sind tief in der VM verankerte Laufzeittests, um z.B. die Einhaltung von Indexgrenzen zu garantieren. Da sich diese selbst auf JVM-Assemblerebene nicht umgehen lassen, ist die Optimierung durch Compiler nur möglich, wenn dafür die eigentliche Java-Idee aufgegeben, und die Programme anstatt in Bytecode in Maschinencode übersetzt werden. Ein Beispiel für diesen Ansatz ist [6]. Immerhin ist dies eine Technologie, die sich javakonform in optimierende JIT-Compiler integrieren ließe.

Der Abstand mit dem die Leistung von Java-Programmen hinter der von Programmen in C/C++ zurückliegt, wird durch bessere JIT-Compiler ständig geringer. Gerade Schleifenprogramme profitieren von dieser Technik, da sie in der Regel kurze Befehlsfolgen besonders oft ausführen, und sich somit gut für eine Optimierung eignen. Zudem bietet Java mit JNI (Java Native Interface) eine Schnittstelle, um Bibliotheken in Maschinencode anzusprechen. Diese können beispielsweise in C/C++ realisiert werden, wodurch sich

[1] `http://www.seti.org` bzw. `http://zap.to/jsetitracker`

bestehende Kommunikationsbibliotheken (etwa MPI) effizient und mit wenig Aufwand nutzen lassen.

Wie in [23] festgestellt wird, sind die parallelen Sprachkonstrukte des Java-Standards für das Erstellen paralleler Programme ausreichend; ein bißchen mehr wäre jedoch besser gewesen. Eine naheliegende Erweiterung ist eine datenparallele Schleife (`forall`), wie sie in einigen hier vorgestellten Projekten realisiert wird. Die Synchronisationsmechanismen von Java sind für das Hochleistungsrechnen nur eingeschränkt verwendbar. So gibt es keine implizite Ordnung der `wait` und `notify` Aufrufe, was für eine direkte Producer/Consumer-Kommunikation jedoch erforderlich wäre. Das Schlüsselwort `synchronized` bietet zwar gegenseitigen Ausschluß, arbeitet jedoch threadbasiert und zeigt in Zusammenhang mit den ebenfalls vorhandenen Verteilungsmechanismen (RMI) ein eher ungewolltes Verhalten. Sinnvolle Ergänzungen sind Synchronisationsklassen für den verteilten Fall, sowie Mechanismen zur kollektiven Kommunikation.

Wenngleich Objektorientierung im Zusammenhang mit Hochleistungsrechnen zumeist unnötigen Mehraufwand bedeutet, bringt sie in Verbindung mit Java auch einige Vorteile. So lassen sich Objekte einfach (z.B. mit JavaParty) von einem Rechner auf einen anderen verschieben - eine Technik, die sich unter Umständen gut für dynamische Probleme oder zur Lastverteilung eignet. Das ausgereifte Laufzeittypsystem von Java erweist sich in der Praxis zudem als überaus nützlich für dynamische Programmierung.

9.2. Erweiterungen zu LooPo

Java ist als Zielsprache für die Parallelisierung von Schleifenprogrammen durchaus geeignet; Zielprogramme in Java-Syntax lassen sich mit relativ geringem Aufwand mit LooPo generieren. Soll Java auch als Quellsprache benutzt werden, so sind vermutlich einige Einschränkungen hinsichtlich der Objektorientierung und der Verwendung von Ausnahmen erforderlich. Um ein Mindestmaß an Komfort zu erhalten, sollte es möglich sein, bestimmte Methoden als "seiteneffektfrei" zu deklarieren; Aufrufe dieser Methoden sollten an beliebigen Stellen im Quellprogramm möglich sein.

Um die erzeugten Programme ausführen zu können ist es erforderlich, einige zusätzliche Funktionen (idealerweise als Klassenbibliothek) zur Verfügung zu stellen. Dazu gehören Methoden zur globalen Synchronisation (`barrier`) und zum Verteilen der Berechnungen (`parfor`). Zudem werden Implementierungen der von LooPo generierten `min`, `max` (etc.) Aufrufe benötigt; da diese beliebig viele Parameter enthalten können, ist nur eine dynamische Implementierung (etwa durch einen Postprozessor) möglich.

Die Verwendung eines Postprozessors für LooPo-Programme empfiehlt sich auch aus anderen Überlegungen. Zum einen ist es so möglich, überflüssige Typkonvertierungen (die in Java relativ teuer sind) nachträglich zu entfernen, zum anderen kann der Rumpf jeder Schleife in eine eigene Methode ausgelagert werden, was modernen JIT-Compilern das Finden und Optimieren von häufig benutzten Codeblöcken (sog. Hot-Spots) erleichtert. Da sich echte Beschleunigung in den betrachteten Beispielen nur durch explizite Kommunikation erreichen lies, sollte die Klassenbibliothek universelle `Send`- und `Receive`-Methoden enthalten, die auf gepufferten Java-Sockets basieren, und deren Aufrufe durch den Postprozessor an die aktuell benutzten Datenstrukturen angepaßt werden.

Sowohl die Klassenbibliothek, als auch der Postprozessor können in Java mit wenig Aufwand realisiert werden. Veränderungen an LooPo sind hierfür (abgesehen von der Ausgabesyntax) zunächst nicht notwendig. Die Methoden zur Verteilung der Berechnungen auf mehrere Rechner, zum Einrichten der Kommunikationsverbindungen sowie zur verteilten Synchronisation können auf Basis von JavaParty relativ einfach realisiert werden. Da JavaParty selbst vollständig in Java realisiert ist, schränkt dies die Zahl der möglichen Zielplattformen nicht ein.

Da die Laufzeit von Java-Programmen in der Regel umgekehrt proportional zu deren Speicherbedarf ist, und es zudem kaum optimierende Java-Compiler gibt, bieten sich zwei weitere Optimierungen an. Wenn es möglich ist, die finale Verwendung einer Variablen (einer Array-Zeile, etc.) zu bestimmen, kann der Speicherplatz in Java explizit durch eine `null`-Zuweisung freigegeben werden. Obwohl aktuelle C/C++ Compiler die komplexen von LooPo erzeugten Programme zufriedenstellend optimieren können, ist dies für die meisten Java-Compiler nicht der Fall. Durch eine explizite Codeoptimierung (Vermeidung von Mehrfachauswertungen, lokale skalare Kopie von read-only Arrayelementen, Vereinfachung von Ausdrücken etc.) sollte sich die Laufzeit deutlich verbessern lassen.

9.3. Perspektiven

Eine Verbindung zwischen kommerziellen und wissenschaftlichen Ansätzen stellen möglicherweise die bereits erwähnten Enterprise-Java-Beans dar. Das Konzept sieht vor, daß der aufzurufende Code, sowie die Schnittstellen für den Aufruf getrennt voneinander in Java realisiert werden. Die dazwischen benötigte Softwareschicht wird automatisch von den Servern automatisch generiert und verbirgt so die interne Struktur der Server vor dem Anwender. Da diese Vorgang für jedes Programm nur einmal benötigt wird, sind an dieser Stelle durchaus auch aufwendige Optimierungen wie etwa eine automatische Parallelisierung denkbar.

Ein sich abzeichnender Trend ist der Einsatz von Clusterrechnern aus Standardkomponenten anstelle von deutlich teureren Spezialrechnern. Die Alpha Linux Cluster Engine (ALiCE) besteht aus 128 Linuxrechnern und löst derzeit eine Connection Machine CM5 in Wuppertal ab. Neben anderen Standards (wie MPI) ist für solche Architekturen automatisch auch Java verfügbar; spezielle Java-Bibliotheken wie die für das in Karlsruhe entwickelte ParaStation Netzwerk, welches auch in ALiCE zum Einsatz kommt, versprechen zudem eine deutliche Senkung der Kommunikationskosten in verteilten Java-Anwendungen.

Durch das hohe Abstraktionsniveau und die einfache Portabilität erschließt Java zweifelsohne neue Bereiche für das Hochleistungsrechnen. Zum einen wird die Entwicklung neuer Techniken vereinfacht, zum anderen ist es möglich, verteilte Programme auf weiträumigen, hybriden Netzwerken (etwa dem Internet) auszuführen. Insgesamt ist Java ein vieldiskutierter und überaus dynamischer Bereich. So darf man in der näheren Zukunft wohl auf eine Vielzahl interessanter Neuerungen gespannt sein.

Literaturverzeichnis

[1] Bryan carpenter. Personal communication.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the Twenty-Fifth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 342–354, January 1998.

[4] G. R. Andrews. *Concurrent Programming, Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1997.

[6] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Apllying high order transformations to Java. In *CPC 2000 Eight International Workshop on Compilers for Parallel Computers*, pages 105–122, January 2000.

[7] M. Baker, B. Carpenter, G. Fox, and S. H. Koo. mpiJava: An object-oriented Java interface to MPI. *Lecture Notes in Computer Science*, 1586:748–762, 1999.

[8] A. Bik and D. Gannon. Exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, 1997.

[9] A. Bik, J. Villacis, and D. Gannon. Experiences with loop parallelization in JAVAR. In *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 1997.

[10] A. Bik, J. Villacis, and D. Gannon. JAVAR: a prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, Nov. 1997.

[11] A. J. C. Bik and D. B. Gannon. javab - a prototype bytecode paralleliation tool. http://www.extreme.indiana.edu/~ajcbik/javab_doc.ps.

[12] Z. Budimlic and K. Kennedy. Optimizing Java—theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, 1997.

[13] D. Caromel and J. Vayssiere. A java framework for seamless sequential, multi-threaded, and distributed programming. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, sep 1998. Special Issue: Java for High-performance Network Computing.

[14] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. *HPJava*: data parallel extensions to Java. *Concurrency: Practice and Experience*, 10(11–13):873–877, Sept. 1998. Special Issue: Java for High-performance Network Computing.

[15] O.-J. Dahl. A note on monitor versions. In *Symposium in Celebration of the Work of C.A.R Hoare*, September 1999. To appear.

[16] N. Ellmenreich, M. Griebl, and C. Lengauer. Applicability of the polytope model to functional programs. In Herbert Kuchen, editor, *Proc. 7th Int. Workshop on Functional and Logic Programming*. Institut für Wirtschaftsinformatik, Westf. Wilhelms-Universität Münster, Apr. 1998.

[17] J. Farley. *Java Distributed Computing*. O´Reilly and Associates, 1998.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[19] M. Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, University of Passau, 1996. Also available as technical report MIP-9701.

[20] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[21] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, Mar. 1997.

[22] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 163–173. IEEE Computer Society Press / ACM Press, 1999.

[23] S. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–632, June 1997.

[24] H. Karl. Bridging the gap between distributed shared memory and message passing. *Concurrency: Practice and Experience*, 10(11–13):887–900, Sept. 1998. Special Issue: Java for High-performance Network Computing.

[25] R. Klemm. Practical guidelines for boosting Java server performance. http://www.cs.ucsb.edu/conferences/java99/papers/21-klemm.ps, 1999.

[26] C. Koelbel, D. Loveman, R. Schreiber, G. S. Jr., and M. Zosel. *THE HIGH PERFORMANCE FORTRAN HANDBOOK*. The MIT Press (Cambridge,London), 1994.

[27] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[29] C. Nester, M. Philippsen, and B. Haumacher. A more efficient rmi for Java. In *ACM 1999 Java Grande Conference*, pages 152–157, June 1999.

[30] S. Oaks and H. Wong. *Java Threads*. O´Reilly and Associates, second edition, 1999.

[31] M. Philippsen and B. Haumacher. More efficient object serialization. *Lecture Notes in Computer Science*, 1586:718–732, 1999.

[32] M. Philippsen and M. Zenger. Javaparty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. http://wwwipd.ira.uka.de/JavaParty/docs/.

[33] W. Pugh. Fixing the java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 89–98. Java Grande Conference, ACM Press, 1999.

[34] A. Spiegel. Pangaea: An automatic distribution front-end for Java. *Lecture Notes in Computer Science*, 1586:93–107, 1999.

[35] N. Yalamanchilli and W. Cohen. Communication performance of java based parallel virtual machines. *Concurrency: Practice and Experience*, 10(11–13):1189–1196, 1998. Special Issue: Java for High-performance Network Computing.

[36] K. Yellick, L. Semenzato, G. Pike, C. Miyamato, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gray, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. pages 825–836, 1998. Concurrency: Practice and Experience, Java Special Issue.

[37] M. Zenger. Transparente Objektverteilung in Java. Studienarbeit, Universität Karlsruhe, http://wwwipd.ira.uka.de/~phlipp/sada/zenger/„ Februar 1997.

[38] G. Zhang, B. Carpenter, G. Fox, X. Li, and Y. Wen. A high level SPMD programming model: HPspmd and its Java language binding. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

[39] H. Zima and B. Chapman. *Supercompilers for Parallel and vector Computers*. ACM Frontier Series. Addison-Wesley, 1991.

Eidesstattliche Erklärung

Hiermit erkläre ich, Andreas Wolf, eidesstattlich, daß ich die Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 4. April 2000