Juggle: Eine verteilte virtuelle Maschine für Java

Michael Schröder, Franz J. Hauck

Universität Erlangen-Nürnberg
Lehrstuhl für Betriebssysteme – IMMD IV
Martensstr. 1, 91058 Erlangen
{schroeder,hauck}@informatik.uni-erlangen.de

Zusammenfassung Die Sprache Java dringt neben World-Wide-Web und Client-Server Anwendungen in immer neue Anwendungsbereiche vor. So werden auch Programme aus dem Bereich Hochleistungsrechnen in Java geschrieben. Für viele Probleme aus diesem Bereich reicht die Leistung eines einzelnen Rechners allerdings nicht aus, es muß deshalb mit Clustern von Rechnern gearbeitet werden. Für den Programmierer bedeutet dies allerdings einen nicht unerheblichen Mehraufwand, da er die Verteilungsaspekte und die unterschiedlichen Semantiken für verteilte Objekte mitberücksichtigen muß.
Das Juggle System bietet hierzu eine Alternative. Juggle implementiert eine verteilte virtuelle Maschine, die transparent für den Benutzer Objekte und Threads auf die beteiligten Rechner verteilt. Eine Codeänderung ist dabei nicht notwendig, so daß auch Programme oder Bibliotheken, für die keine Quellen erhältlich sind, verteilt ablaufen können. Durch eine geeignete Instrumentierung wird ständig zur Laufzeit die optimale Position für Objekt und Threads bestimmt und über Migrationen und Replikationen umgesetzt.

1 Einleitung

Die Programmiersprache Java hat sich in den letzten Jahren als Sprache für das Internet etabliert. Dies umfaßt neben Applets im World-Wide-Web auch Client-Server Anwendungen, die über standardisierte Schnittstellen wie RMI [Sun97] oder CORBA [OMG98] kommunizieren.

Es stellt sich die Frage, ob Java auch für die Lösung von Problemen aus den Naturwissenschaften geeignet ist, wo die Anwendungen sich durch extrem hohe Anforderungen an die Rechenleistung der Systeme auszeichnen (sogenanntes *High Performance Computing, HPC*).

Java bietet dem Programmierer Objektorientierung, Sprachkonstrukte für Threadprogrammierung und hohe Portabilität. Allerdings wurde Java als interpretierte Sprache entwickelt und kann auch trotz neuer *Just-in-Time* Compiler noch nicht mit der Leistung von Fortran- oder C-Compilern mithalten. Um diesen Leistungsverlust aufzufangen und darüber hinaus die Leistung vieler Recheneinheiten zu nutzen, ist der Einsatz einer verteilten Applikation, die auf einem Cluster von Rechnern läuft, naheliegend.

Die Programmierung einer solchen verteilten Applikation ist allerdings nicht trivial, da sich der Programmierer zusätzliche Gedanken über die Verteilung der Objekte auf die Rechnerknoten, die Verteilung der Threads und die Aspekte der Netzwerkschicht (Verhalten bei Netzproblemen, usw.) machen muß. Eine optimale Verteilung der Threads wird außerdem dadurch erschwert, daß Methodenaufrufe an entfernte Objekte ein "Auswandern" der Threads bewirken.

Ideal wäre eine verteilte Ablaufumgebung (*Java Virtual Machine*, JVM), die automatisch Objekte und Threads migriert und für eine optimale Auslastung der Knoten sorgt. Das *Juggle* System realisiert eine solche verteilte JVM. Mit Juggle kann eine beliebige parallele Java-Anwendung auf mehreren Rechnern verteilt ablaufen. Die virtuelle Maschine wurde instrumentiert, so daß während des Programmlaufes ständig günstige Positionen für Objekte und Threads errechnet werden. Die errechneten Konfigurationen werden über Migrationen und Replikationen umgesetzt.

Mittlerweile gibt es Werkzeuge wie '`javab`' [BiGa97] und '`javar`' [BiGa97a] aus dem *High Performance Java* Projekt, die ein sequentielles Programm in ein nebenläufiges umwandeln, so daß auch sequentielle Programme von der Verteilung auf mehrere Rechner profitieren können.

2 Das Design von Juggle

Bei Programmiermodellen für das Hochleistungsrechnen können zwei wesentliche Kategorien unterschieden werden: Systeme mit verteiltem gemeinsamen Speicher (Distributed Shared Memory, DSM) und Systeme, die mit Hilfe von Nachrichten oder entfernten Prozeduraufrufen kommunizieren. Im ersten Fall sind die Aktivitätsträger (Threads) über die beteiligten Rechner verteilt. Diese greifen auf gemeinsame Daten zu. Das Speichersubsystem verteilt meist auf der Basis von Speicherseiten die Daten auf die Rechner, auf denen Zugriffe stattfinden. Die Datenkonsistenz ist vom Speichersubsystem zu gewährleisten und wird aus Effizienzgründen oft aufgeweicht. Die Nachteile dieses Ansatzes sind zum einen die möglicherweise unglückliche Plazierung von Datenobjekten auf dieselbe Speicherseite. Zum anderen sind die Threads in solchen Systemen in der Regel vom Anwender statisch zu verteilen. Eine automatische Lastverteilung ist nicht möglich.

Bei Prozedur- oder Methodenaufrufen springt der aufrufende Aktivitätsträger konzeptionell an den Ort des Zielobjekts und kehrt dann an den Ausgangspunkt zurück. In der Implementierung bearbeitet im verteilten Fall jedoch ein anderer jeweils lokaler Thread den Aufruf. Ein Lastverteilungsalgorithmus steht daher vor dem Problem, daß seine Ergebnisse nur für eine kurze Zeitspanne Gültigkeit besitzen, da Threads blockiert werden und auf Ergebnisse eines Aufrufs warten oder neue Threads Aufrufe bearbeiten. Ähnlich verhält es sich bei reiner Nachrichtenkommunikation, wenn diese blockierend durchgeführt wird. Die Verteilung von Daten muß in diesem Programmiermodell immer durch den Anwender erfolgen.

Die statische Verteilung von Daten und Threads zu den Knoten hat zusätzliche Nachteile:

- Die Verteilung hängt von der eingesetzten Hardware ab (Kommunikationsnetzwerk, Prozessoren, Speicherausstattung) und muß daher für jede Konfiguration neu angepaßt werden. Ein NUMA Supercomputer ist nicht mit einem Workstationcluster vergleichbar.
- Eine hochwertige Verteilung wird oft durch die Eingabedaten beeinflußt. Moderne adaptive Verfahren erschweren zudem die Korrelation der Daten und der Zugriffsmuster.
- Viele Algorithmen arbeiten in Phasen, die jeweils andere Verteilung erfordern.
- Das Ändern eines Algorithmus kann großen Einfluß auf die nötige Verteilung haben. Die statische Verteilung muß dann ebenfalls geändert werden.

Juggle beschreitet daher einen anderen Weg. Das verwendete Programmiermodell ist das von Java, bei dem Threads Bestandteil der Sprache sind. Juggle besteht aus einer verteilten virtuellen Java Maschine, die auf allen beteiligten Rechnerknoten läuft und die für eine automatische Lastverteilung sorgt. Da Juggle selbst für die Verteilung der Objekte und Threads sorgt, kann jedes vorhandene parallele Java-Programm ohne Änderungen verteilt ablaufen. Daher können auch Programme und Bibliotheken, für die kein Quellcode zur Verfügung steht, von der Leistungsfähigkeit eines Hochleistungsrechners oder Compute-Clusters profitieren. Die Verteilung wird immer auf die augenblickliche Konfiguration der Juggle-Maschinen hin optimiert.

Ähnlich wie in DSM-Systemen ist jedes Java-Objekt in Juggle von jedem beteiligten Knoten aus ansprechbar. Im Gegensatz zu DSM-Systemen werden die gesamten Objektdaten aber nicht notwendigerweise zu dem zugreifenden Knoten verschickt. Stattdessen wird bei einem Zugriff auf eine Instanzvariable das entsprechende Datum von dem Knoten angefordert, auf dem die Objektdaten im Moment gespeichert werden. In Juggle können daher die Daten eines Objekts transparent migriert werden.

Da viele Objekte nur einmal modifiziert werden und danach nur noch lesend auf sie zugegriffen wird (dies gilt insbesondere für die Objekte, welche die Eingabedaten repräsentieren), macht es Sinn, solche Objekte nicht zu migrieren sondern zu replizieren. So kann jeder Knoten seine eigene Kopie der Objekte benutzen und muß nicht zeitaufwendige Zugriffe auf andere Knoten durchführen. Wird auf ein repliziertes Objekt schreibend zugegriffen, werden zuerst alle Replikas vernichtet bevor der Zugriff durchgeführt wird.

Beim Aufruf von Methoden wird im Gegensatz zu gängigen verteilten Objektsystemen kein entfernter Aufruf durchgeführt. Stattdessen wird der Methodencode lokal ausgeführt, unabhängig davon, ob die entsprechenden Objektdaten lokal liegen oder nicht. Allein der Zugriff auf die Instanzvariablen (bzw. die Instanzvariablen des Klassenobjekts) führt zu einem Nachrichtenaustausch. Der Programmcode wird dafür auf alle beteiligten Knoten repliziert (Code braucht normalerweise deutlich weniger Speicher als die Daten). Der Vorteil dieses Ansatzes ist, daß die Position eines Threads weder vom abgearbeitetem Code noch von

den zu bearbeitenden Methoden und der Verteilung der zugehörigen Objekten abhängt.

Die Verteilung von Objektdaten und Threads kann nun beliebig erfolgen und wird von Juggle in folgender Weise dynamisch und automatisch vorgenommen: Juggle optimiert die Verteilung der Objekte, indem es ein Maß für die Last der Threads auf die Objekte bestimmt. Hierbei werden für jedes Objekt die Anzahl der Zugriffe aller Threads in einer bestimmten Zeitspanne gezählt. Diese Anzahl ist ein Maß für die Last der Threads auf das Objekt. Es wird immer zu dem Thread migriert, der die größte Last erzeugt, da dieser Thread am meisten von einem lokalen Objekt profitiert.

Mit diesem Verfahren wird die Anzahl der Zugriffe auf entfernte Knoten stark reduziert. Unabhängig davon wird periodisch die Auslastung der Knoten überprüft und gegebenenfalls korrigiert. Dabei wird allerdings auch die Korrelation zwischen den Objekten und den Threads berücksichtigt: Threads, die auf die gleichen Objekte Last ausüben, werden möglichst auf den gleichen Knoten bewegt. Es wird also sowohl die Last auf die Knoten balanciert als auch die Anzahl der entfernten Objektzugriffe minimiert.

3 Implementierung der Juggle JVM

Die Juggle JVM ist eine völlige Neuentwicklung, da sich so die verschiedenen Objektverteilungsalgorithmen am besten integrieren ließen. Juggle wurde in Standard ANSI-C programmiert. Als Thread Bibliothek wurden POSIX Threads eingesetzt, um einfache Portierbarkeit zu gewährleisten. Die Juggle JVM bietet gegenüber einer nur lokal ablaufenden virtuellen Maschine einige spezielle Erweiterungen, die in den folgenden Abschnitten näher erläutert werden sollen.

3.1 Zugriffe auf entfernte Knoten

Einige Bytecodes greifen auf den Datenbereich von Objekten zu und können daher Interaktion mit anderen Knoten bewirken:

- `getfield`, `getstatic`, `putfield`, `putstatic` (Zugriffe auf Instanz- und Klassenvariablen)
- `?aload`, `?astore` (Zugriffe auf Arrays)
- `monitorenter`, `monitorexit` und Aufruf (Rücksprung) von `synchronized` Methoden.

Wenn die virtuelle Maschine einen dieser Bytecodes bearbeitet, muß überprüft werden, ob das angesprochene Objekt lokal oder auf einem entfernten Knoten liegt. Trifft letzteres zu, wird eine Nachricht zu dem betroffenen Knoten geschickt und der Thread solange blockiert bis die Antwort eingetroffen ist.

Einige *native* Methoden wie `arraycopy` und EA-Operationen können ebenfalls einen Nachrichtenaustausch bewirken.

Tritt ein Kommunikationsproblem auf (weil zum Beispiel einer der Knoten nicht erreichbar ist), bricht Juggle mit einer Fehlermeldung ab.

3.2 Instrumentierung

Da die virtuelle Maschine für jedes Objekt herausfinden muß, welcher Knoten die größte Last auf es ausübt, werden bei jedem Objekt folgende Zusatzinformationen gehalten:

- ein Zeiger auf eine *node*-Struktur, die den Knoten beschreibt, auf dem sich das Objekt befindet. Ist das Objekt lokal, ist dieser Zeiger `NULL`.
- ein Zähler um die Zugriffe von lokalen Threads zu zählen (`lcnt`).
- ein Zähler für alle Fernzugriffe (`rcnt`).

Jeder Zugriff auf das Objekt bewirkt die Erhöhung von `lcnt` oder `rcnt`. Dabei reicht die Ermittlung der Größenordnungen aus, so daß es nicht nötig ist, die Zähler durch ein Lock zu sichern. Erreicht die Summe beider Zähler einen konfigurierbaren Schwellwert, wird die Position des Objektes neu bestimmt und gegebenenfalls durch Migration oder Replikation angepaßt.

Wenn die Anzahlt der Zugriffe von lokalen Threads die der entfernten überwiegt (`lcnt >= rcnt`), bleibt das Objekt auf dem Knoten, auf dem es sich befindet. Andernfalls ist es vielleicht sinnvoll, das Objekt auf einen anderen Knoten zu verlagern. Zur Ermittlung dieses Knotens reicht ein einzelner Zähler allerdings nicht aus, da die einzelnen Knoten unterschieden werden müssen. Daher wird das Objekt um einen Zähler für jeden Knoten erweitert. Dies wird erst im Bedarfsfall gemacht, da auf eine sehr große Anzahl von Objekten nur lokal zugegriffen wird und so signifikant Speicherplatz gespart werden kann.

3.3 Threadmigration

Für die Threadmigration müssen weiterhin die Verwandschaftsbeziehungen der Threads ermittelt werden. Der Grad der Verwandschaft ist dabei umso höher, je öfter die Threads auf die gleichen Objekte zugreifen. Hierzu wird ein Samplingalgorithmus eingesetzt: Jedes Objekt wird um ein zusätzliches Feld erweitert, in dem die `ThreadID` des Threads gespeichert wird, der zuletzt auf das Objekt zugegriffen hat. Jeder Thread führt ein Feld von Zählern mit, das mit der `ThreadID` indiziert wird. Bei jedem Zugriff wird der Zähler für die im Objekt gespeicherte `ThreadID` erhöht und dann die eigene `ThreadID` im Objekt gespeichert.

Juggle überprüft periodisch die Positionen der Threads auf den Knoten. Dabei geht als Randbedingung ein, daß alle Knoten möglichst gleichmäßig mit Threads belegt werden. Bei der Vergabe der Threads an die Knoten werden die in jedem Thread gespeicherten und durch den zuvor beschriebenen Algorithmus ermittelten Verwandschaftsbeziehungen berücksichtigt, so daß die Anzahl der Nachrichten mitunter deutlich gesenkt werden kann.

3.4 Objektreplikation

Juggle repliziert ein Objekt anstelle es zu migrieren, wenn die Anzahl der Lesezugriffe nach dem letzten Schreibzugriff “genügend” hoch ist. Hierfür wird

ein weiterer Zähler verwendet, der bei jedem Lesezugriff erhöht und bei einem Schreibzugriff gelöscht wird. Soll ein Objekt migriert werden, wird überprüft, ob der Zähler einen konfigurierbaren Wert überschreitet. Ist dies der Fall, wird das Objekt repliziert.

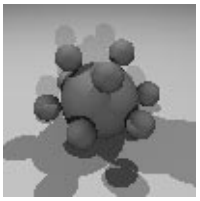
Das System merkt sich pro Objekt über ein Bitfeld, welche Knoten Replikas besitzen. Wird ein Schreibzugriff durchgeführt, werden zuerst alle diese Knoten benachrichtigt und so die Replikas dort gelöscht.

4 Meßwerte

Die Juggle JVM besteht im Moment aus ca. 8400 Zeilen ANSI-C Code. Sie ist in etwa so schnell wie Suns Java-1.0 Version. Es spricht nichts dagegen, die Geschwindigkeit durch Integration eines *Just-in-Time* Compilers zu erhöhen.

Als Testprogramm wurde ein einfacher Raytracer implementiert (ca. 1300 Zeilen Java Code), der das von Eric Haines entwickelte "*Neutral File Format*" [Hain87] lesen kann. Damit kann der Raytracer die Testszenen aus Haines "*Standart Procedural Database*" Paket bearbeiten, die häufig zum Testen von Raytracern eingesetzt werden.

Für die Messungen wählten wir den "Balls" Datensatz, der aus einer Ebene, 10 Kugeln und drei Lichtquellen besteht. Um die Messungen einfach zu halten, verzichteten wir auf Spiegeleffekte. Die Szene wurde in vier Quadrate unterteilt, die jeweils von einem Thread berechnet wurden. Als Rechencluster wurden vier Sun Ultra-1 Workstations verwendet. Dies bedeutet eine gleichmäßige Verteilung der Threads auf die Knoten, ohne daß Threadmigrationen stattfinden.



Balls Szene

10 Kugeln

1 Ebene

3 Lichquellen

Wir verwendeten für die Messungen drei Versionen von Juggle. Die einfachste Variante benutzt eine feste Zuweisung von Objekten an die Threads und damit an die Knoten. Die Position eines Objektes wird bei seiner Erzeugung festgelegt: es wird die Position des Erzeugerthreads verwendet. Jeder Zugriff auf Objekte, die von anderen Threads erzeugt worden sind, verursachen daher den Austausch von Nachrichten.

Die zweite Version arbeitet mit Objektmigration. Die Objekte, welche die Szenenbeschreibung repräsentieren und das Objekt, daß das errechnete Bild speichert, wandert von Knoten zu Knoten, um die Anzahl der Fernzugriffe zu verringern.

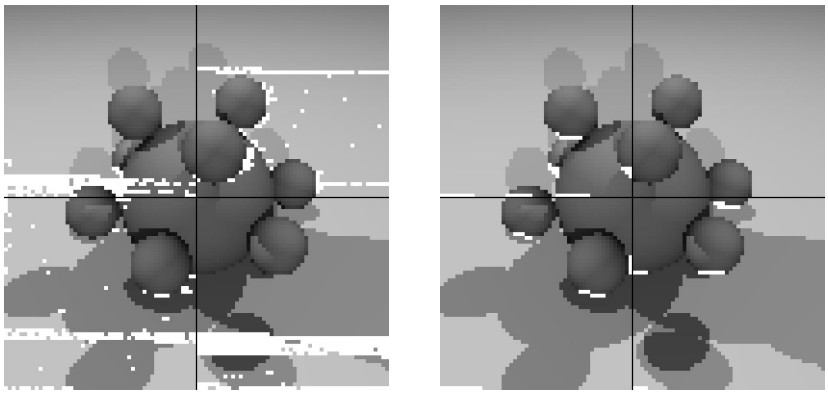
Schließlich wurde in der dritten Version auch Replikation erlaubt, um die Berechnung weiter zu beschleunigen. Die Meßwerte aller drei Berechnungen sind in Tabelle 1 zusammengefaßt:

Tabelle1. Ergebnisse für die "Balls"-Szene

	feste Position	mit Migrationen	mit Migrationen und Replikationen
Objekte		101331	
verteilte Objekte		188	
Migrationen		3302	54
Replikationen			251
lokale Zugriffe	$6,9 \cdot 10^6$	$12 \cdot 10^6$	$12 \cdot 10^6$
Fernzugriffe	$5,5 \cdot 10^6$	$0,41 \cdot 10^6$	$0,02 \cdot 10^6$

Die Meßwerte zeigen deutlich die Vorteile von Migrationen und Replikationen auf. Die Anzahl der Fernzugriffe konnte um einen Faktor von ca. 200 verringert werden.

Die Wirkungen der Migrationen und Replikationen auf die Objekte des Raytracers können in einfacher Form graphisch wiedergegeben werden. In den Bildern in Abbildung 1 ist die Aufteilung der Berechnung in die vier Bereiche zu sehen, die von den einzelnen Threads berechnet werden. Dabei wurden die Bildpunkte nur dann dargestellt, wenn das Objekt lokal lag, das für die Helligkeit des Pixels verantwortlich war (das sogenannte *Shader*-Objekt). Da das Bild zeilenweise von unten nach oben und jede Zeile von links nach rechts berechnet wird, spiegelt das resultierende Bild auch die Bewegung der Objekte über die Zeit wieder.



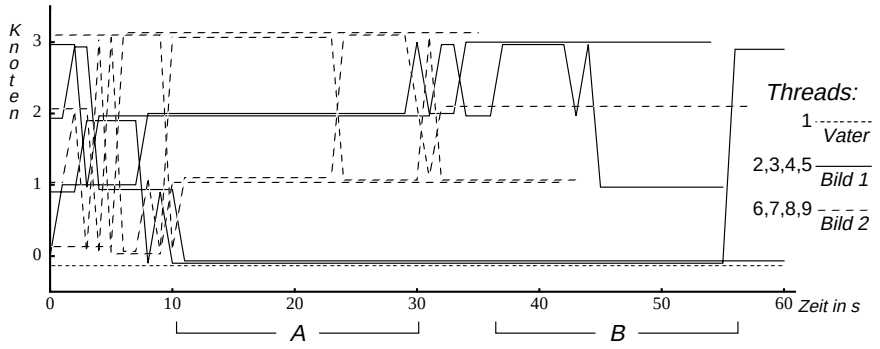
Berechnung mit Migrationen mit Migrationen und Replikationen

Abbildung1. Bildpunkte mit lokalen Shader-Objekten in der "Balls" Szene

Im linken Bild war Objektmigration erlaubt. Man kann erkennen, daß die Shader der Kugeln, die nur in einem Bildausschnitt liegen und daher nur von einem Thread angesprochen werden, schnell zu dem zugehörigen Knoten wandern. Andererseits springen Objekte wie die Ebene, die von allen Threads angesprochen wird, von Zeit zu Zeit auf andere Knoten.

Erlaubt man auch Replikationen ergibt sich das rechte Bild. Die Shaderobjekte werden schnell auf die Knoten repliziert, da bei der Bildberechnung auf ihre Daten nur lesend zugegriffen wird.

Zur Veranschaulichung der Threadmigrationen wurde der Raytracer erweitert, so daß zwei Bilder gleichzeitig berechnet und in einem Gesamtbild gespeichert werden. Für die Tests wurden 8 Threads eingesetzt, d.h. vier Threads pro Bild. Folgende Graphik gibt die Position der Threads auf den vier Rechenknoten wieder:



Thread 1 ist der Vaterthread der Arbeitsthreads. Seine einzige Aufgabe ist es, die Szenen einzulesen und die 8 Arbeiter zu starten. Danach wartet er, bis alle Thread fertiggerechnet haben und schreibt dann das Ergebnisbild. Während der Berechnung trägt er nicht zur Auslastung der Knoten bei und bleibt daher auf Knoten 0. Die anderen Threads werden durch den Threadmigrationsalgorithmus nach kurzer Zeit so verteilt, daß Threads, die das gleiche Bild berechnen, auf den gleichen Knoten arbeiten (Bereich A). Wenn einzelne Threads fertig gerechnet haben, werden die restlichen Threads so umverteilt, daß möglichst viele Knoten rechnen (Bereich B).

4.1 Speedupmessungen

Um Speedupmessungen durchzuführen benutzten wir eine etwas komplexere Variante der Balls-Szene mit 91 reflektierenden Kugeln:

Anzahl der Knoten	1	2	4
Dauer der Berechnung	120s	75s	56s
Speedup	–	1,6	2,14

Bei dem verwendeten Workstation-Cluster beträgt die durchschnittliche Dauer zum Senden und Empfangen einer Nachricht ca. zwei Millisekunden. Im Falle

von vier Knoten verschickt jeder Knoten ca. 6000 Nachrichten, so daß allein für den synchronen Nachrichtenaustausch 12 Sekunden verbraucht werden.

Als zweites Testprogramm wurde ein Glättungsalgorithmus implementiert. Er glättet ein quadratisches Feld von Größen indem er jedes Element durch den Mittelwert seiner vier Nachbarn ersetzt. Für die Parallelisierung wurde das Feld in Streifen geteilt, die parallel geglättet werden. Dabei greift die Berechnung der Ränder eines Streifens auf den Nachbarstreifen zu. Abbildung 2 zeigt einen Plot einer Startmatrix und die zugehörige geglättete Matrix.

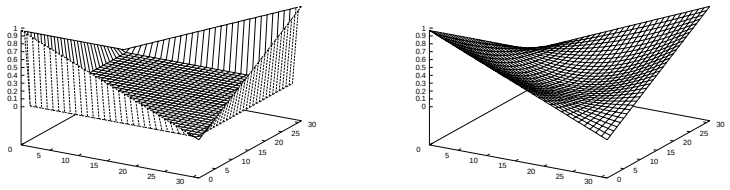


Abbildung2. Eingangsmatrix und Ergebnis des Glättungsalgorithmus

Für die Messungen wurden 8 Threads eingesetzt, die eine 800×800 Matrix in 100 Schritten geglättet haben. Die Meßwerte sind in folgender Tabelle dargestellt:

Anzahl der Knoten	1	2	4	8
Dauer der Berechnung	520s	341s	208s	143s
Speedup	–	1,53	2,5	3,64

Die Rechenleistung bricht bei 8 Knoten deutlich ein, was sich durch Rechenzeitverlust beim Warten in einer globalen Barriere des Programmes nach jedem Glättungsschritt erklären läßt.

Insgesamt läßt sich sagen, daß die Leistung von Juggle nicht an durch den Programmierer verteilte Programme (z.B. über MPI) heranreicht. Dafür mußte an den Javaprogramm nichts geändert werden, um sie auf Juggle zum Laufen zu bringen.

5 Verwandte Arbeiten

Da Java eigens für den Einsatz in Netzwerken und dem WWW konzipiert wurde, beschäftigten sich schon frühere Arbeiten mit der Verteilung von Java-Objekten. Bei den meisten dieser Arbeiten muß der Entwickler sich selbst um die Verteilung von Threads und Daten kümmern.

Java RMI, Suns Remote Method Invocation [Sun97] erlaubt es, Methoden entfernter Java-Objekte aufzurufen. Die Syntax eines Aufrufes unterscheidet sich nicht von der bei lokalen Objekten. Realisiert wird dies durch Proxy- und Skeletonklassen, die automatisch erzeugt werden können. Für den Benutzer ergeben sich allerdings einige Unterschiede zu normalen Aufrufen:

- Ein verteiltes Objekt muß von einer speziellen Remote-Basisklasse abgeleitet werden. Weiterhin kann jeder Aufruf an ein verteiltes Objekt eine RemoteException verursachen, was häufig zu vielen `try-catch` Blöcken führt.
- Ein verteiltes Objekt kann nur über sein remote Interface angesprochen werden, daher müssen spezielle Methoden geschrieben werden, um auf Instanzvariablen oder statische Methoden zugreifen zu können.
- Es ist daher nicht möglich, Zugriffsbeschränkungen zu verwenden (`private` / `protected`).
- Javas Synchronisationsprimitive funktionieren nicht mit verteilten Objekten.
- Lokale Objekte werden als Kopie übergeben, alle verteilten Objekte per Referenz.

JavaParty [PhZe97] versucht viele der Einschränkungen von RMI aufzuheben. Eine verteilte Klasse wird hier durch ein zusätzliches Schlüsselwort "`remote`" bei der Klassendefinition gekennzeichnet. Ein Precompiler erzeugt daraus normalen Java Code, der über RMI mit den verteilten JVMs kommuniziert. Wie bei lokalen Objekten können Instanzvariablen gelesen und verändert werden. Objekte können nach der Erzeugung zu anderen Knoten migriert werden oder direkt auf anderen Knoten angelegt werden. Es gibt allerdings einige Schwachpunkte:

- RMIs Probleme mit Javas Synchronisationsprimitiven gelten auch für JavaParty.
- Die Objektmigration geschieht nur applikationsgesteuert.
- Der statische Datenanteil einer Klasse kann nicht migriert werden.
- Jede remote Klasse wird durch den Precompiler zu **zehn** neuen Klassen umgesetzt.

Ähnlich wie JavaParty arbeiten *Do!* [LaPa98] und *Manta* [NMB+99] mit einem Präprozessor, der Zugriffe auf verteilte Klassen in RMI-Aufrufe überführt. Bei Do! sind keine Zugriffe auf Instanzvariablen erlaubt. Manta benutzt verschiedene Transportprotokolle, um die Kommunikation in einem weitverteilten System zu optimieren.

Auch *Remote Objects in Java* [NaSr96] bietet die Erzeugung verteilter Objekte an. Anstelle einer Methode wird aber ein neues Schlüsselwort "`remotenew`" verwendet, das einen neuen Opcode erzeugt. Es wird daher ein eigener Compiler und eine eigene JVM benötigt. Die Parameter bei Aufrufen an verteilten Objekten dürfen nur primitive Datentypen, also keine Objektreferenzen enthalten.

Anders als die auf Message Passing basierten Systeme beruht *Java/DSM* [YuCo97] auf einem zugrundeliegenden DSM-System (*Distributed Shared Memory*). Dieses ermöglicht direkte Speicherzugriffe auf gemeinsame Speicherbereiche.

Java/DSM benötigt eine eigene JVM, die den gemeinsamen Speicher für die Datenbereich der Java-Objekte benutzt und die Garbage Collection performant durchführen kann.

Hyperion [MMH98] bietet wie Juggle dem Benutzer die Sicht einer einzigen verteilten virtuellen Maschine. Hierbei wird der Java Code durch einen Präcompiler in C-Code umgewandelt. Jedes Objekt besitzt eine feste Zuordnung zu einem Knoten (es gibt also keine Objektmigration), wird aber bei Zugriffen von anderen Knoten grundsätzlich repliziert. Schreibzugriffe auf ein Objekt werden dabei erst zum Hauptknoten des Objekts übertragen, wenn ein Monitor betreten oder verlassen wird. Weiterhin werden beim Betreten eines Monitors grundsätzlich alle Replikas vernichtet, um einen konsistenten Blick auf die Instanzvariablen zu gewährleisten. Dies könnte bei Programmen, die viel mit Monitoren arbeiten, zu größeren Performanceproblemen führen. Hyperion bietet keine Möglichkeit der Threadmigration, da die feste Zuordnung von Objekten zu Knoten die Threads auch an die (lokalen) Objekte bindet.

Orca [BKT92] ist vom Aufbau am ehesten mit Juggle vergleichbar. In der Programmiersprache Orca beschriebenen *Distributed shared objects* werden entweder auf alle beteiligten Knoten repliziert oder auf genau einem Knoten plaziert, wobei diese Entscheidung dynamisch zur Laufzeit geändert werden kann. Die Grundlage für die Entscheidung bilden jedoch auch Ergebnisse aus einer Flußanalyse zur Compilezeit [BaKa93]. Im Gegensatz zu Juggle wird für den Fall eines nicht-lokalen Zugriffs ein entfernter Aufruf vorgenommen, während in Juggle der Thread lokal weiterarbeitet und lediglich die zugegriffenen Instanzvariablen übertragen werden. Außerdem kennt Orca keine Threadmigration.

6 Zusammenfassung

Juggle bietet dem Anwender die Abstraktion einer verteilten virtuellen Maschine. Um ein Programm auf mehreren Rechnern verteilt laufen zu lassen, ist also keine Codeänderung notwendig. Dies ist eine entscheidende Entlastung für den Programmierer, der sich so keine Gedanken über die Rechnerarchitekturen machen muß. Die Programme laufen unverändert auf Workstationclustern oder NUMA Supercomputern. Außerdem wird so der Einsatz von Funktionen aus Bibliotheken ermöglicht, für die keine Quellen verfügbar sind. Juggle erweitert also den Slogan von Java “write once, run everywhere” um die Dimension der Rechnercluster.

Literatur

[BaKa93] Bal, H.E.; Kaashoek, M.F.: Object distribution in Orca using compile-time and run-time techniques. *OOPSLA'93 Conference, SIGPLAN Notices*, 28(10), Oct. 1993, pp. 162-177.

[BiGa97] Bik, A.J.C.; Gannon, D.B.: Exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6), Jun. 1997.

[BiGa97a] Bik, A.J.C.; Villacis, J.E.; Gannon, D.B.: javar: a prototype Java restructuring compiler. *Concurrency, Practice and Experience*, 9(11), Nov. 1997.

[BKT92] Bal, H.E.; Kaashoek, M.F.; Tanenbaum, A.S.: Orca: a language for parallel programming of distributed systems. *IEEE Trans. on Software Eng.*, 18(3), March 1992, pp. 190-205.

[Hain87] Haines, E.: A proposal for standard graphics environments, *IEEE Computer Graphics and Applications*, 7(11), Nov. 1987.

[LaPa98] Launay, P.; Pazat, J.-L.: *Generation of distributed parallel Java programs.* Publication interne no. 1171, Feb. 1998.

[MMH98] MacBeth, M.W.; McGuigan K.A.; Hatcher, Ph.H.: Executing Java threads in parallel in a distributed-memory environment. *IBM Centre for Advanced Studies Conference, Toronto, Canada*, Nov.-Dez. 1998.

[NaSr96] Nagaratnam N.; Srinivasan, A.: Remote Objects in Java. *IASTED Intl. Conf. on Networks*, Jan. 1996.

[NMB+99] van Nieuwpoort, R.; Maassen, J.; Bal, H.E.; Kielmann, T.; Veldema, R.: Wide-area parallel computing in Java. *ACM Java Grande Conf.*, June 1999.

[OMG98] Object Management Group: *The common object request broker: architecture and specification.* Rev. 2.2, OMG Doc. formal/98-02-01, Feb. 1998.

[PhZe97] Philippsen, M.; Zenger, M.: JavaParty: transparent remote objects in Java. *Concurrency: Practice & Experience*, 9(11), Nov. 1997.

[Sun97] Sun Microsystems Comp. Corp.: *Java remote method invocation specification.* Rev. 1.4, JDK 1.1, 1997.

[LiYe97] Lindholm, T; Yellin, F.: *The Java virtual machine specification.* Addison-Wesley, 1997.

[YuCo97] Yu, W.; Cox, A.: Java/DSM: a platform for heterogenous computing. *Concurrency: Practice & Experience*, 9(11), Nov. 1997.