

Juggle: Eine verteilte virtuelle Maschine für Java

M. Schröder

Mai 1998

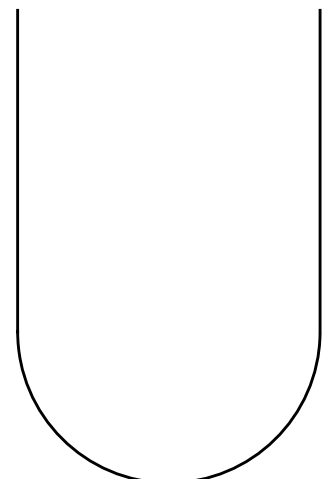
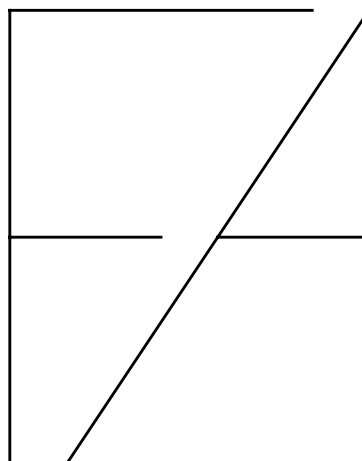
TR-14-3-98

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



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

Mai 1998

Kurzfassung

Die Sprache Java dringt neben World-Wide-Web und Client-Server Anwendungen in immer neue Anwendungsbereiche vor. So werden schon Programme aus dem Bereich Hochleistungsrechnen in Java geschrieben. Für viele Probleme aus diesem Bereich reicht die Leistung eines einzelnen Rechners allerdings nicht aus, deshalb muß 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 [OMG95] kommunizieren.

In letzter Zeit wird verstärkt untersucht, wie sich Java als Programmiersprache für Probleme aus den Naturwissenschaften eignet. Diese Anwendungen zeichnen sich durch extrem hohe Anforderungen an die Rechenleistung der Systeme aus (sogenanntes *High Performance Computing*,

HPC). Java besticht durch seine Objektorientierung und die integrierte Unterstützung von mehreren Aktivitätsträgern (*Threads*), da hierdurch sauber programmierte Anwendungen geschrieben werden können, die moderne Multiprozessorsysteme ausnützen können.

Allerdings reicht die Rechenleistung eines einzelnen Rechners für *HPC*-Anwendungen häufig nicht aus. Erschwerend kommt hinzu, daß Java als interpretierte Sprache entwickelt wurde und die Leistung neuer *Just-in-Time* Compiler noch nicht mit der von Fortran- oder C-Compilern mithalten kann. Daher 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 Anwendung auf mehreren Rechnern verteilt ablaufen. Die virtuelle Maschine wurde so instrumentiert, daß während des Programmlaufes ständig die optimalen Positionen für Objekte und *Threads* bestimmt und über Migrationen und Replikationen realisiert werden.

Mittlerweile gibt es schon Werkzeuge wie 'javab' [BikGa97] und 'javar' [BikGa97a] aus dem *High Performance Java* Projekt, die ein sequentielles Programm parallelisieren, so daß auch solche Programme von der Verteilung auf mehrere Rechner profitieren können.

2 Das Design von Juggle

Juggle besteht aus einer verteilten virtuellen Maschine, die auf jedem der beteiligten Knoten gestartet wird und die für die automatische Lastverteilung sorgt. Da *Juggle* eigenständig die *Threads* und Objekte auf die beteiligten Knoten verteilt, kann der Benutzer vorhandene parallele Programme ohne Änderungen starten. Somit können auch Programme und Bibliotheken, für die keine Quellen verfügbar sind, von der Rechenleistung eines Workstationclusters profitieren.

Juggle benötigt keine statische Verteilungskonfiguration. Der Gebrauch einer solchen wurde verworfen, da er mit einigen Nachteilen verbunden ist:

- Die Verteilung ist abhängig von der vorhandenen Hardware (Netzwerke, Prozessoren, Speicher) und muß daher für jede Konfiguration neu erstellt werden. Ein NUMA Rechner verhält sich zum Beispiel grundlegend anders als ein Workstationcluster.
- Die optimale Verteilung ist häufig abhängig von den Eingabedaten. Bei modernen adaptiven Verfahren wird erst während der Berechnung klar, welche Bereiche der Eingabedaten für den Hauptteil der Rechenzeit verantwortlich sind. Die optimale Abbildung der Daten auf Objekte ist also am Anfang gar nicht bekannt.

- Viele Programme arbeiten mit mehreren Berechnungsphasen, die andere Anforderungen an die Verteilung stellen. Jede Phase kann eine andere optimale Verteilung besitzen, was eine statische Konfiguration erschwert.
- Eine Codeänderung kann sich stark auf die optimale Verteilung auswirken. Ein “Experimentieren” mit Algorithmen, wie es gerade in der Forschung erstrebenswert ist, ist also mit zusätzlicher Arbeit verbunden.

Aus diesen Gründen berechnet Juggle zur Laufzeit ständig die optimale Verteilung für Objekte und Threads. Stimmt die aktuelle Position eines Objektes nicht mit der neu berechneten überein, wird das Objekt zur neuen Position migriert. Die Verteilung der Threads geschieht so, daß die Last auf die beteiligten Knoten möglichst gleichmäßig ist und gleichzeitig die Interaktionen zwischen den Knoten gering gehalten werden.

Da viele Objekte (zum Beispiel Objekte, die Eingabedaten repräsentieren) nur einmal geschrieben und dann nur noch ausgelesen werden, bietet sich für sie an, die Migration durch Replikation zu ersetzen. Jeder Knoten kann dabei eine eigene Kopie der Objekte besitzen, um unnötige Zugriffe auf entfernte Knoten zu vermeiden. Wird auf ein repliziertes Objekt schreibend zugegriffen, müssen erst die Kopien auf den anderen Knoten vernichtet werden.

Für die gleichmäßige Auslastung der Knoten sind einige neue Konzepte nötig. Bei Systemen, die auf dem verteilten Methodenaufruf (*remote method invocation, RMI*) basieren, wandern die Threads durch die Aufrufe zu Objekten auf anderen Knoten. Es werden also Objekte verteilt und für die Verteilung werden die Interaktionen der Objekte untereinander berücksichtigt.

Juggle verfolgt einen völlig anderen Ansatz. Der Code des zu bearbeitenden Programmes wird auf alle beteiligten Rechner repliziert. Ein Methodenaufruf an ein entferntes Objekt führt zu keiner Interaktion mit dem entfernten Knoten, da der Code lokal vorhanden ist. Allein der Zugriff auf Instanzvariablen bzw. auf Variablen des Klassenobjektes bewirkt eine Interaktion. Es wird kein Code der Applikation auf dem entfernten Knoten abgearbeitet, stattdessen wird lediglich das angesprochene Datum transportiert.

Dies hat einen entscheidenden Vorteil: es macht wieder Sinn, von der Position eines Threads und der Verteilung von Threads zu reden. Wie wird nun bestimmt, wie die optimale Verteilung der Threads und der Objekte aussieht? Juggle benutzt hierzu das Konzept der Last auf ein Objekt. Betrachtet man ein Objekt, so kann man für einen Zeitabschnitt messen, wie oft und von welchen Threads auf das Objekt zugegriffen wurde. Dieser Wert stellt eine Art Last dar, die jeder Thread auf das Objekt erzeugt. Für jedes Objekt ergibt sich dadurch eine optimale Position, nämlich die desjenigen Threads, dessen Last auf es am größten ist.

Durch diesen ersten Schritt können also die optimalen Positionen für alle Objekte bezüglich der Threads ermittelt werden. Als zweiter Schritt müssen die Threads auf die Knoten des Systems verteilt werden. In die Berechnung fließen wieder die Lastwerte ein, denn neben der Auslastung der Knoten soll die Kommunikation minimiert werden. Dies geschieht, indem Threads, die auf dieselben Objekte eine hohe Last ausüben, möglichst auf die gleichen Knoten verteilt werden.

3 Realisierung der Juggle JVM

Die Juggle JVM wurde komplett neu entwickelt, um freie Hand bei der Entwicklung der Objektverteilung zu haben. Als Programmiersprache wurde "C" gewählt, als portable Threadbibliothek werden POSIX Threads eingesetzt.

3.1 Zugriffe auf entfernte Knoten

Folgende Bytecodes greifen auf Objekte zu und können daher zu einer Interaktion mit einem anderen Knoten führen:

- `getfield`, `getstatic`, `putfield`, `putstatic` (Zugriffe auf Instanz- und Klassenvariablen)
- `?aload`, `?astore` (Zugriffe auf Arrays)
- `arraylength` (das Resultat kann aber lokal gespeichert werden, da die Arraygröße in Java eine Konstante ist)
- `monitorenter`, `monitorexit` sowie Aufrufe von (und Rücksprünge aus) `synchronized` Methoden

Bei jedem dieser Opcodes wird überprüft, ob das zugehörige Objekt (bei `get/putstatic` das Klassenobjekt) auf einem anderen Knoten liegt. Ist dies der Fall, wird eine Nachricht mit dem Opcode und den Parametern verschickt und der aktuelle Thread solange blockiert, bis eine Antwort eingetroffen ist. Bei Fehlern in der Übertragung (wenn zum Beispiel der Zielknoten nicht erreichbar ist) wird das Programm mit einer Fehlermeldung beendet.

3.2 Instrumentierung

Für jedes Objekt muß bestimmt werden, welcher Thread die größte Last auf es ausübt. Zu diesem Zweck wird die Objektstruktur um vier Elemente erweitert:

- `thread_id pos`; Dies ist die ID des Threads, zu dem das Objekt im Moment zugeordnet ist.
- `int lcnt`; Ein Zähler für die Zugriffe des lokalen Threads (also IDpos).
- `int rcnt`; Ein Zähler für die Zugriffe aller anderen Threads.
- `acnts_t *acnts`; Ein Verweis auf eine Struktur, die weitere Informationen zur Verteilung enthält.

Jeder Zugriff auf das Objekt (also durch die obigen Bytecodes) erhöht entweder `lcnt` oder `rcnt`. Ein Verlust von einigen Zugriffen ist nicht relevant, daher müssen die Zähler nicht durch ein Lock gesichert werden. Nach jeweils 100 Zugriffen ($lcnt + rcnt \geq 100$) wird über die Zuordnung des Objekts zu den Threads entschieden (und damit über die Objektmigration):

Wenn genügend Zugriffe des "lokalen" Threads erfolgten ($lcnt + 20 \leq rcnt$), bleibt das Objekt weiterhin dem Thread mit ID `pos` zugeordnet.

Andernfalls reicht ein einzelner Zähler für die Zugriffe der anderen Threads nicht aus, da die Threads einzeln überprüft werden müssen. Dies geschieht über die zusätzliche Struktur `acnts_t`. Da eine große Anzahl von Objekten nur von einem Thread angesprochen wird und daher diese Struktur nicht benötigt, wird sie, um Speicherplatz zu sparen, nur im Bedarfsfall angelegt.

Da potentiell beliebig viele Threads erzeugt werden können, ist es zu aufwendig, für jeden Thread einen Zähler bereitzustellen. Stattdessen wird nur ein Zähler verwendet, der über eine Samplingstrategie nacheinander die Zugriffe aller Threads mißt.

Die Struktur `acnts_t` enthält zu diesem Zweck folgende Elemente:

- `thread_id cur;` Die ID des Threads, der im Moment gezählt wird.
- `int curcnt;` Der Zähler für Thread `cur`.
- `thread_id next;` Die ID des Threads, der als Nächstes gezählt werden soll
- `thread_id bst;` Die ID des Threads mit den meisten Zugriffen (bis jetzt).
- `int bstcnt;` Das aktuelle Maximum.

Der Algorithmus arbeitet folgendermaßen: Jeder Zugriff des Threads mit der ID `cur` erhöht den Zähler `curcnt`. Bei anderen Threads wird überprüft, ob die ThreadID größer als `cur` aber kleiner als `next` ist. Ist dies der Fall wird die ID nach `next` geschrieben. Dadurch wird das Minimum aller ThreadIDs größer als `cur` gebildet.

Nach 100 Zugriffen wird überprüft, ob `curcnt` größer als `bstcnt` ist, also ob ein neues Maximum gefunden wurde. Ist dies der Fall, wird `bst` und `bstcnt` aktualisiert. Weiterhin wird `curcnt` gelöscht und `cur` auf `next`, also auf die nächste ThreadID gesetzt.

Ist keine nächste ThreadID gefunden worden (`next == 0`), wurden alle Threads überprüft. Der Thread mit ID `bst` hat die meisten Zugriffe ausgeübt und das Objekt wird daher diesem Thread zugeschlagen.

3.3 Replikation

Ein Objekt wird zur Replikation freigegeben, wenn über eine “genügend lange” Zeit keine Schreibzugriffe erfolgt sind. Die Struktur `acnts_t` wird durch ein zusätzliches Element erweitert:

- `int nowrites;` Anzahl der Zyklen in denen nicht geschrieben wurde

Jeder schreibende Zugriff setzt diesen Zähler wieder auf Null zurück. Nach 100 Zugriffen wird der Zähler erhöht. Soll ein Objekt migriert werden, wird überprüft, ob `nowrites` einen genügend hohen Wert enthält. In diesem Fall wird das Objekt repliziert.

4 Messungen

Die Juggle JVM besteht im Moment aus ca. 4600 Zeilen C-Code. Sie ist etwa so schnell wie Suns java-1.0 Version, aber es spricht nichts dagegen, Juggle um einen *Just-in-Time* Compiler zu erweitern. Im Moment wird die Verteilung auf andere Knoten nur simuliert, an einer Verteilung auf andere Rechner wird gearbeitet.

Für die Tests haben wir einen kleinen Raytracer implementiert (ca. 1300 Zeilen Java Code), der das von Eric Haines entwickelte "*Neutral File Format*" lesen kann. Damit kann der Raytracer die Testszenen aus Haines "*Standard Procedural Database*" [Hain87] Paket bearbeiten, welche oft zum Testen von Raytracern eingesetzt wird.

Als Testszene wählten wir den "Balls" Datensatz. Er besteht aus einer Ebene, 10 Kugeln und drei Lichtquellen. Um die Ergebnisbilder einfach zu halten, wurde die Reflexion der Kugeln ausgeschaltet. Für die Simulation unterteilten wir das Bild in vier Quadrate, die jeweils von einem Thread berechnet wurden. Jeder Thread sollte dabei auf einem eigenen Knoten laufen.

Wir starteten das Programm mit drei unterschiedlichen Versionen von Juggle. Die einfachste Variante benutzt feste Positionen für die Objekte. Wird ein neues Objekt erzeugt, ist damit die Position festgelegt, nämlich bei dem Thread, der das Objekt angelegt hat. Sämtliche Zugriffe von anderen Threads erfordern damit Interaktionen zwischen den Knoten.

Als zweiten Test erlaubten wir Objektmigration. Die Objekte der Szenenbeschreibung und das Objekt, daß die Ergebnisdaten hält, wandern daher von Knoten zu Knoten und erhöhen so die Anzahl der lokalen Zugriffe.

Für die dritte Variante durften Objekte auch repliziert werden. Die Meßwerte für die drei Tests sind in der folgenden Tabelle zusammengefaßt.

	feste Objektpositionen	mit Migration	mit Migration und Replikation
Objekte	101331		
verteilte Objekte	188		
Migrationen		6959	169
Replikationen			175
Lokale Zugriffe	$6,9 \cdot 10^6$	$9,4 \cdot 10^6$	$12 \cdot 10^6$
Zugriffe von anderen Threads	$5,5 \cdot 10^6$	$2,8 \cdot 10^6$	$0,22 \cdot 10^6$

Tabelle 1 Ergebnisse für die "Balls" Szene

Man kann an den Werten für die Zugriffe gut den positiven Effekt von Migration und Replikation erkennen. Die Zugriffe von anderen Threads konnten damit um einen Faktor von mehr als 20 verringert werden.

Die Migrationen und Replikationen der Objekte lassen sich graphisch darstellen. Das linke Bild in Abbildung 1 zeigt die Bereiche, welche die vier Threads abarbeiten. Jedem der Threads wurde eine Schraffur zugeordnet. Wird bei der Berechnung der Bildpunkte festgehalten, welchem Thread das Objekt zugeordnet ist, das für die Farbe des getroffenen Gegenstandes verantwortlich ist (das *Shaderobjekt*), so kann jeder Punkt mit einer Schraffur versehen werden. Da die Bereiche zeilenweise von unten nach oben und in den Zeilen von links nach rechts berechnet werden, ist so auch zeitliche Verlauf erkennbar.

Das mittlere Bild gibt die Positionen der Shaderobjekte wieder, wenn nur mit Migration gearbeitet wird. Man kann erkennen, daß der Shader für Gegenstände, die ausschließlich in dem Bereich eines einzelnen Threads sind, schnell zu dem zugehörigen Thread wandert. Andererseits wird auf die Ebene unter den Kugeln fortlaufend von allen Threads aus zugegriffen, deshalb springt sie von einem Thread zum nächsten.

Im rechten Bild ist auch Replikation erlaubt gewesen. Hier sieht man, daß die Shaderobjekte nach kurzer Zeit repliziert werden, da auf sie nur noch lesend zugegriffen wird.

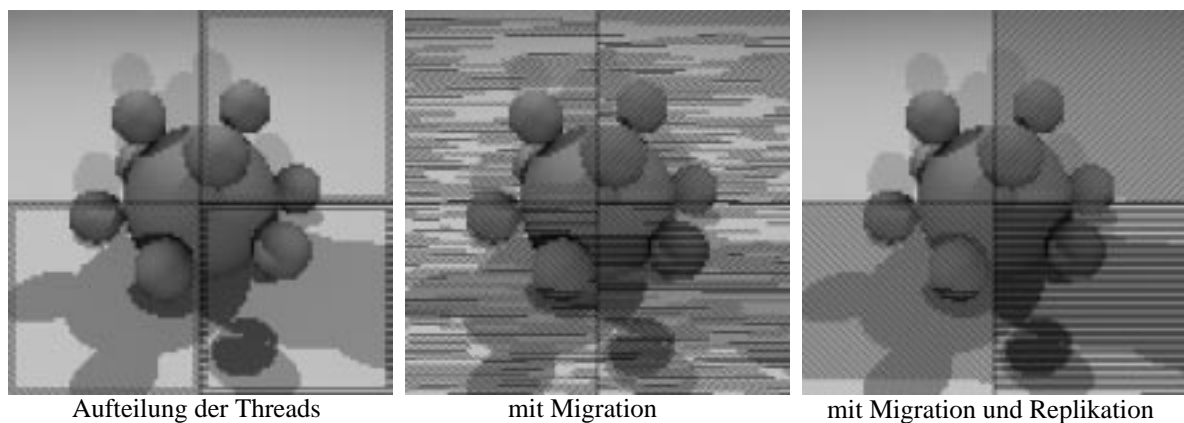


Abb. 1 Positionen der Shaderobjekte in der “Balls” Szene

5 Verwandte Arbeiten

Da Java eigens für den Einsatz in Netzwerken und dem WWW konzipiert wurde, beschäftigten sich schon mehrere Arbeiten mit der Verteilung von Objekten.

- *Suns Remote Method Invocation*

Das RMI Paket [Sun97] erlaubt es, Methoden entfernter 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 Remote-Exception verursachen, was häufig zu vielen try-catch Blöcken führt.

- Ein verteiltes Objekt kann nur über sein remote Interface angesprochen werden, daher können statische Methoden oder private Methoden nicht verwendet werden.
 - Auf Instanzvariablen eines verteilten Objekts kann nicht zugegriffen werden.
 - Javas Synchronisationsprimitive funktionieren nicht mit verteilten Objekten.
 - Lokale Objekte werden als Kopie übergeben. Alle verteilten Objekte per Referenz.
- *JavaParty*

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 kleinere Schwachpunkte:

 - RMIs Probleme mit Javas Synchronisationsprimitiven gelten auch für JavaParty.
 - Es können zwar Knoten dynamisch zu dem System hinzugenommen werden, aber das Entfernen von Knoten ist nicht möglich.
 - Die Objektmigration geschieht nur applikationsgesteuert.
 - Der statische Datenanteil einer Klasse kann nicht migriert werden.
 - Jede remote Klasse wird durch den Precompiler zu 10 Klassen umgesetzt.
 - *Do!*

Das *Do!* Projekt [LauPa98] benutzt wie JavaParty einen Preprozessor, der Zugriffe auf verteilte Klassen auf RMI-Aufrufe abbildet. Eine Klasse wird als verteilt deklariert, indem sie das Interface Accessible anbietet. Über eine remoteNew Methode können Objekte auf anderen Knoten erzeugt werden. *Do!* erlaubt im Gegensatz zu JavaParty keine Zugriffe auf Instanzvariablen verteilter Objekte.
 - *Remote Objects in Java*

Auch Remote Objects [NaSr96] in Java bietet die Erzeugung verteilter Objekte an. Anstelle einer Methode wird aber ein neues Schlüsselwort “remotew” 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.
 - *Java/DSM*

Anders als die Message Passing basierten Systeme beruht Java/DSM [YuCox97] auf einem zugrundeliegenden DSM (*distributed shared memory*) System. Dieses ermöglicht direkte Speicherzugriffe auf gemeinsame Speicherbereiche. Java/DSM benötigt eine eigene JVM, die den gemeinsamen Speicher benutzt und die Garbage Collection performant durchführen kann.

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.

Durch die interne Instrumentierung der virtuellen Maschine ist Juggle in der Lage, während des Programmlaufes ständig die optimalen Positionen für Objekte und Threads zu berechnen und die Verteilung durch Migrationen und Replikationen anzupassen.

7 Literatur

- BikGa97 Bik, A.J.C.; Gannon, D.B.: Exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6), Jun. 1997
- BikGa97a Bik, A.J.C.; Villacis, J.E.; Gannon, D.B.: javar: a prototype Java restructuring compiler, *Concurrency, Practice and Experience*, 9(11), Nov. 1997
- Hain87 Haines, E.: A Proposal for Standard Graphics Environments, In *IEEE Computer Graphics and Applications*, v. 7, no. 11, Nov. 1987
- LauPa98 Launay, P.; Pazat, J.-L.: *Generation of distributed parallel Java programs*, Publication interne no. 1171, Feb. 1998
- NaSr96 Nagaratnam N., Srinivasan, A.: Remote Objects in Java, In *IASTED Intl. Conf. on Networks*, Jan. 1996
- OMG95 Object Management Group; *The Common Object Request Broker: Architecture and Specification*, 2.0 ed, 1995
- PhZe97 Philippsen, M.; Zenger, M.: JavaParty - Transparent Remote Objects in Java, In *Concurrency: Practice & Experience*, Vol. 9, Nr. 11, Nov. 1997
- Sun97 Sun Microsystems Comp. Corp.: *Java Remote Method Invocation Specification*, Revision 1.4, JDK 1.1, 1997
- LiYe97 Lindholm, T; Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley, 1997
- YuCox97 Yu, W.; Cox, A.: Java/DSM: A Platform for Heterogenous Computing, In *Concurrency: Practice & Experience*, Vol. 9, Nr. 11, Nov. 1997