JavaParty – portables paralleles und verteiltes Programmieren in Java

Michael Philippsen, Matthias Zenger und Matthias Jacob

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5, 76128 Karlsruhe
phlipp@ira.uka.de
http://wwwipd.ira.uka.de/JavaParty/

Zusammenfassung Während Java Threads (Aktivitätsstränge) als geeignetes Sprachmittel für die Programmierung von SMPs (Parallelrechnern mit gemeinsamem Speicher) anbietet, fehlen elegante und ausreichende Sprachmittel für die Programmierung von Parallelrechnern mit verteiltem Speicher (DMPs), also auch für Cluster von Arbeitsplatzrechnern. Die in der Java-Distribution angebotene explizite Socket-Kommunikation und der Aufruf entfernter Methoden (RMI) erfordern bei der Portierung eines für eine SMP-Maschine entwickelten mehrsträngigen Programms auf eine DMP-Maschine erhebliche Programmänderungen und -erweiterungen.
JavaParty behebt diesen Mißstand, ermöglicht ein Java-artiges Programmieren auch von DMPs und Clustern von Arbeitsplatzrechnern und verallgemeinert die Idee des plattformunabhängigen Codes auch für Parallelrechner unterschiedlicher Architekturen. Die Erweiterung beruht auf einem neuen, die Klassendeklaration ergänzenden, Klassenmodifikator `remote`, mit dem potentiell entfernt zu realisierende Objekte für den JavaParty-Übersetzer gekennzeichnet werden. Aus dem Attribut wird dann portabler Java-Code sowohl für SMPs als auch DMPs erzeugt, ohne den Programmierer der Komplexität von expliziter Socket-Kommunikation bzw. RMI auszusetzen.

1 Einleitung

Zwei wichtige Gründe für Javas [6] Popularität sind die Verfügbarkeit von portablen Programmierschnittstellen für den Zugriff auf die Internet-Kommunikationsmechanismen [7] und das Vorhandensein von Threads und Synchronisierung im Sprachumfang [16].

Java ermöglicht damit einerseits die Erstellung von mehrsträngigen Programmen, die auf SMPs echt parallel ausgeführt werden können. Andererseits sind die vorhandenen Kommunikationsbibliotheken, insbesondere der entfernte Methodenaufruf (RMI) [24], für die Erstellung von klassischen Client-Server-Anwendungen umfassend und ausreichend.

Ähnlich wir Corba erlaubt es RMI, Methoden von Objekten aufzurufen, die auf einem anderen Rechner instantiiert sind. Diese Objekte sind dazu vom Programmierer auf der Server-Seite bei einem „Name-Server“ zu registrieren; der

Client kann dann eine Referenz auf das entfernte Objekt erhalten und dieses fortan weitgehend wie ein gewöhnliches Java-Objekt verwenden. RMI ist für instabile Netzwerkverbindungen und Client-Server-Architekturen entworfen worden; diese Sicht ist beim Programmieren deutlich zu spüren und unterscheidet sich wesentlich von dem Paradigma eines gemeinsamen Speichers. Beispielsweise bietet RMI eine reichhaltige Sammlung von Ausnahmebedingungen an (Exceptions), die der Programmierer nutzen kann bzw. muß, um seine Anwendung vor eventuell auftretenden Netzproblemen zu schützen.

Neben RMI sind systemnahe Sockets in Java verwendbar, mit deren Hilfe der Programmierer eigene Kommunikationsprotokolle auf unterster Ebene realisieren und dadurch wesentlich effizientere Kopplungen der nebenläufigen Programme erreichen kann, als dies durch RMI möglich ist. Jedoch sind Programme mit expliziter Socket-Kommunikation schon allein wegen des erforderlichen Protokollentwurfs erheblich schwerer zu durchschauen und zu warten.

Kurz gesagt: Java bietet zwar adäquate Mechanismen an, um mittel- und grobgranulare Parallelität auf SMPs und grobgranulare Client-Server-Applikationen mit geringem Kommunikationsaufwand auf verteilten Systemen auszudrücken. Jedoch bietet Java weder Unterstützung für Cluster von Arbeitsplatzrechnern, deren Kommunikationsnetze gewöhnlichen LANs und WANs in puncto Latenzzeit, Durchsatz und Verläßlichkeit deutlich überlegen sind [1, 2, 25], noch für irreguläre parallele Anwendungen, die einen höheren Kommunikationsbedarf aufweisen.

JavaParty erweitert den vom Programmierer zu beherrschenden Java-Sprachumfang so wenig wie möglich. Mit Hilfe eines Laufzeitsystems und einer pures Java erzeugenden Vorverarbeitung wird das parallele Programmieren von Clustern von Arbeitsplatzrechnern ermöglicht und damit Portabilität zwischen SMPs und DMPs erreicht.

Erstens ist JavaParty also eine Programmierumgebung, die zur Realisierung von Cluster-Anwendungen verwendet werden kann. Derzeit wird in Kooperation mit dem Stanford Exploration Project [3] die Nutzbarkeit von JavaParty für datenintensive geophysikalische Anwendungen untersucht [22]. Im Rahmen der DFG-Forschergruppe RESH [23] sind Projekte zum „Data-Mining“ und zur Realzeit-Verfolgung von Fahrzeugen in Verkehrsszenen angelaufen. Zweitens bildet JavaParty die Grundlage von Forschungsarbeiten, deren Ziele die Lokalitätsoptimierung und damit die Reduktion der Kommunikationszeiten sind. Zur Laufzeit, durch statische Analyse im Übersetzer oder durch vom Programmierer zu liefernde Annotationen wird derzeit versucht, Informationen über bestehende Lokalitätsbeziehungen zwischen Objekten und Threads zu identifizieren, um die Objekte passend zu plazieren bzw. zu migrieren. Drittens werden Servlets zur Verringerung der Web-Server-Belastung mit Hilfe von JavaParty programmiertechnisch wesentlich leichter verteilt, als dies mit CGI und Perl möglich ist.

In Abschnitt 2 werden zunächst die wesentlichen Eigenschaften von JavaParty vorgestellt. Um die Vorteile des JavaParty-Ansatzes zu demonstrieren, schließt sich daran ein detaillierter Vergleich von JavaParty-Programmen mit solchen an, die entweder Sockets oder RMI explizit verwenden. Der vierte Abschnitt präsen-

tiert die Transformation von JavaParty-Code in puren Java-Code. Kurze Blicke auf die gegenwärtig mit JavaParty erreichbaren Laufzeiten in Abschnitt 5 und auf verwandte Arbeiten in Abschnitt 6 beschließen diesen Beitrag.

2 JavaParty im Überblick

Ein mehrsträngiges Java-Programm kann ohne wesentliche Programmüberarbeitungen leicht in ein verteiltes JavaParty-Programm fast gleicher Quellcodegröße verwandelt werden, indem diejenigen Klassen und Threads identifiziert werden, die über die verteilte Umgebung verstreut werden sollen. Der Programmierer ergänzt die Klassendeklarationen der betroffenen Klassen um den Klassenmodifikator `remote`, den JavaParty neu einführt. Dieser Modifikator ist die einzige Erweiterung von Java.[1] Da die Threads in Java ebenfalls als Objekte realisiert sind, kann auf dieselbe Weise eine Klasse von Threads erzeugt werden, die auf entfernten Knoten ausgeführt werden.

JavaParty realisiert einen gemeinsamen Adreßraum: Obwohl die Objekte auf verschiedenen Knoten plaziert sind, können deren Methoden und Variablen (sowohl statische als auch Instanz-Bestandteile) genauso aufgerufen bzw. angesprochen werden wie von Java gewöhnt. JavaParty verbirgt Adressierungsfragen und Kommunikation mit entfernten Knoten vor dem Programmierer, behandelt eventuell auftretende Netzwerkprobleme intern und erfordert weder Entwurf noch Implementierung von speziellen Kommunikationsprotokollen.

Abgesehen von der Unterscheidung zwischen den als `remote` gekennzeichneten Objekten und gewöhnlichen Java-Objekten ist die Objektplazierung für den Programmierer transparent. Entfernte Objekte und Threads brauchen nicht „per Hand“ auf bestimmte Knoten verteilt zu werden, weil diese Aufgabe vom Übersetzer und vom Laufzeitsystem übernommen werden kann, die sich mit Lokalitätsfragen und der Optimierung der Kommunikationsoperationen befassen. Dazu bietet das JavaParty-System Verteilungsstrategien an, die konsultiert werden, ehe ein neues entferntes Objekt bzw. ein Thread erzeugt wird. Die ihrerseits ebenfalls in Java implementierten Verteilungsstrategien sind mit Hilfe des Entwurfsmusters „Strategie“ [5] an das Laufzeitsystem angekoppelt und dadurch zur Laufzeit auswähl- und wechselbar. Der erfahrene Programmierer kann JavaParty sogar um problemspezifische Verteilungsstrategien erweitern. Ferner können Objekte migrieren, um Zugriffslokalität zu erreichen. Neben dem Übersetzer und dem Laufzeitsystem kann auch der Programmierer selbst Migrationsaufträge erteilen.

3 JavaParty im Vergleich zu Sockets und RMI

Wir haben einige der Salishan-Probleme [4] viermal implementiert: in Java mit Threads, in Java mit expliziter Socket-Kommunikation, in Java mit RMI und

[1] Es wurde experimentell ferner eine `forall`-Anweisung zum datenparellelen Programmieren ergänzt, die jedoch hier nicht weiter untersucht wird. Der interessierte Leser sei auf [21] verwiesen.

in JavaParty. Unter den Problemen waren die sortierte Berechnung aller Isomere der Paraffinmoleküle (ohne Wiederholungen), die Simulation eines Wartezimmers einer ärztlichen Gemeinschaftspraxis und Hammings Problem. Zum Verständnis der folgenden Diskussion ist kein tiefes Verständnis dieser Probleme erforderlich; es ist ausreichend zu wissen, daß sie irregulär, schwer vorhersagbar und deutlich verschieden von gewöhnlichen numerischen Anwendungen sind.

Während die Laufzeiten der vier Implementierungen keine signifikanten Unterschiede aufwiesen, zeigten die Programme sehr unterschiedliche Quellcode-Größen.

	Java Sockets	RMI	Java-Party
`wc -l`	1277 2086	2123	1277
	63.3%	66.2%	
`sdiff`	0 992	969	28
	77.7%	75.9%	2.2%

Die normalen Java-Programme und die entsprechenden JavaParty-Programme haben die gleiche Anzahl von Zeilen (1.277). Im Vergleich dazu wächst für die auf Sockets basierende Version die Code-Größe um 63.3% auf 2.086 Zeilen an. Obwohl man ein geringeres Wachstum erwarten würde, benötigt die RMI-Version sogar 66.2% mehr Code, also 2.133 Zeilen, obwohl die Erstellung und Überprüfung der Socket-Version im Vergleich zur RMI-Version nahezu doppelt so lange gedauert hat.

Diese Ergebnisse werden noch deutlicher, wenn nicht die absolute Code-Größe berücksichtigt wird, sondern diejenigen Zeilen gezählt werden, die geändert, ergänzt und gelöscht wurden. Diese Zeilenzahl ist in der unteren Hälfte der Tabelle (siehe `sdiff`[2]) angegeben.

Im folgenden diskutieren wir die Nachteile der Socket- und der RMI-Version im einzelnen und stellen dadurch die spezifischen Vorteile von JavaParty heraus.

Programmstruktur und Erzeugung entfernter Objekte. Weil JavaParty die Illusion eines gemeinsamen Adreßraums erzeugt, sind anders als bei den Socket- und RMI-Versionen weder eine künstliche Auftrennung der Applikation in Client und Server erforderlich, noch ist komplizierter Code nötig, um entfernte Objekte zu erzeugen.

Im allgemeinen folgen sowohl Socket- also auch RMI-Programme dem Client-Server-Ansatz. Der Programmierer muß Client- und Server-Anteile in seiner Applikation identifizieren und diese auf das zugrundeliegende Netzwerk abbilden. Nehmen wir für den Moment an, daß eine derartige Aufteilung konzeptuell leicht zu erzeugen ist. Dann muß der Programmierer zumindest zwei unterschiedliche Programme schreiben und manuell starten, jeweils eines für den Server und einer für den Client. Alternativ kann ein Initialisierungs-Skript erstellt werden, das den

[2] Die Werte wurden mit Hilfe von `sdiff -sb file1.java file2.java | egrep -c '[<|>]'` ermittelt.

manuellen Programmstart vereinfacht. Die Aufteilung in zwei unterschiedliche Programme verursacht ein moderates Code-Wachstum.

Ist die Aufteilung in Client- und Server-Portionen nicht offensichtlich, verursacht die Verwendung von Sockets oder RMI deutliche Mehrarbeit, weil es beide Mechanismen nicht erlauben, entfernte Objete zu erzeugen. Für die Socket-Lösung muß das Kommunikationsprotokoll Pakete enthalten, die der Empfängerprozeß in Konstruktoraufrufe umsetzt. Analog für RMI: Auf der Maschine, auf der das entfernte Objekt erzeugt werden soll, muß ein Hilfsobjekt instantiiert sein, dessen Methoden entfernt aufgerufen werden können. Eine dieser Methoden ruft ihrerseits den Konstruktor des zu erzeugenden Objekts auf und liefert einen Zeiger auf dieses Objekt als Rückgabewert an den Aufrufer zurück. Diese indirekte entfernte Objekterzeugung, die einen Agenten auf der Zielmaschine zum Aufruf des passenden Konstruktors benötigt, ist fehleranfällig und verursacht lästiges Code-Wachstum.

Verbindungsaufbau. Da JavaParty Kommunikations- und Adressierungsmechanismen vor dem Programmierer verbirgt und mit etwaigen Netzwerkproblemen intern umgeht, bleibt der Code deutlich kleiner als bei äquivalenten Socket- und RMI-Implementierungen.

Um eine Verbindung zwischen Client und Server herzustellen, muß der Programmierer sowohl in Socket- als auch in RMI-Programmen TCP/IP-Adressierungsaufgaben bearbeiten: Er muß den IP-Namen der ausführenden Maschine zur Laufzeit herausfinden, und er muß über die verwendeten Port-Nummern und die textuellen Namen informiert sein, die der RMI-„Name-Server“ entfernten Objekten zugeordnet hat.

Das folgende Code-Fragment zeigt, was bei der Implementierung mit Hilfe von Sockets auf Seite des Clients zum Verbindungsaufbau erforderlich ist. Es ist dem Programmierer überlassen, wie er mit `IOException`s umgeht, die z.B. auf besetzte Ports zurückzuführen sind oder von einem Server verursacht werden, der noch nicht dazu gekommen ist, die geforderte Verbindung anzubieten.

```
DataInputStream is;
DataOutputStream os;
try {
  Socket mySocket = new Socket(server, port);
  is = new DataInputStream(mySocket.
           getInputStream());
  os = new DataOutputStream(mySocket.
           getOutputStream());
} catch (IOException e) {
  ... what to do?
  ... Try again? Involve the user?
}
```

Bei der RMI-Lösung muß der Server bei dem „Name-Server“ (rmiregistry) des eigenen Rechners registriert sein. Ein solcher „Name-Server“ muß auf allen Knoten installiert sein, auf denen sich Objekte befinden, die von außen angesprochen werden sollen.

```
// create server
Server server = null;
try {
  server = new Server(...);
} catch (RemoteException e) {
  ...what to do?
}
// register server
try {
  InetAddress iaddr = InetAddress.getLocalHost();
  String url = "//"+iaddr.getHostName()+"/server";
  java.rmi.Naming.bind(url, server);
  server.work();
} catch (AlreadyBoundException e) {
  ... what to do?
} catch (MalformedURLException e) {
  ... what to do?
} catch (java.rmi.UnknownHostException e) {
  ... what to do?
} catch (java.net.UnknownHostException e) {
  ... what to do?
} catch (RemoteException e) {
  ... what to do?
}
```

Der obige Code ist bzgl. Maschinennamen noch unvollständig. Auf der Server-Seite ist ähnlicher Code ebenfalls zum Beenden erforderlich, d.h. um Objekte beim „Name-Server“ abzumelden. Auch auf der Client-Seite ist eine analoge Kaskade von `catch`-Befehlen erforderlich, wenn der Client versucht, vom „Name-Server“ eine entfernte Referenz auf dem Server zu erhalten.

Der Programmierer hat also an mindestens drei Stellen mit diversen Ausnahmebedingungen und dem „Name-Server“ zu kämpfen, um ein Objekt entfernt verwenden zu können. Während diese drei Stellen evtl. noch hinter einer vereinfachenden Schnittstelle verborgen werden können, ist dies beim Methodenaufruf (s.u.) nicht mehr ohne starke Programmveränderungen möglich.

Während `RemoteExceptions` in WANs auftreten, sind sie in Parallelrechnern und eng gekoppelten Netzen von Arbeitsplatzrechnern unwahrscheinlich, außer sie werden durch Socket- oder RMI-Mechanismen selbst verursacht. Während unserer Programmerstellung traten derartige Ausnahmebedingungen auf; sie waren aber stets darauf zurückzuführen, daß wir Adressierungsfehler gemacht haben. Kurz gesagt: Ein Programmierer, der sich nicht mit IP-Namen, Port-Nummern und URLs herumschlagen muß, macht in diesem Bereich auch keine Fehler.

In Abschnitt 4 zeigen wir, wie JavaParty einen transparenten Verbindungsaufbau realisiert.

Kommunikation/Methodenaufruf. In JavaParty sind erstens Entwurf und Implementierung eines Kommunikationsprotokolls unnötig. Zweitens bleiben Signaturen existierender Methoden unverändert, wenn ein Java-Programm in ein JavaParty-Programm umgewandelt wird; insbesondere muß der Programmierer sich nicht um Netzwerk-Ausnahmebedingungen kümmern. Beide Eigenschaften

führen dazu, daß der gegebene Java-Code nur unwesentlich überarbeitet werden muß.

Es ist offensichtlich, daß die Socket-Implementierung dies nicht leistet. Stattdessen muß der Programmierer ein verklemmungsfreies Kommunikationsprotokoll konzipieren, ein passendes Leitungsprotokoll zur Festlegung der Botschaftenformate definieren und Empfängerprozesse implementieren.

Erstaunlicher ist, daß auch bei der Verwendung von RMI sich nur wenig dieser Komplexität eingesparen läßt. Da in RMI nicht auf Instanzvariablen von entfernten Objekten zugegriffen werden kann, muß der Programmierer bei der Klassendefinition spezielle Zugriffsfunktionen vorsehen und diese anstelle der Variablenzugriffe verwenden. Für Arrays ist eine ganze Kollektion solcher Zugriffsfunktionen erforderlich, z.B. um auf einzelne Dimensionen des Arrays zugreifen zu können. Darüberhinaus kann RMI auch nicht für statische Methoden und statische Variablen verwendet werden. Diese Einschränkungen können zu erheblichen Code-Veränderungen und -Ergänzungen führen.

Neben diesen durch Restriktionen von RMI bedingten Code-Veränderungen sind folgende Code-Veränderungen inhärent durch RMI gefordert:

- Für jedes entfernt zu verwendende Objekt muß ein Interface deklariert werden, das die entfernt aufrufbaren Methoden ausdrücklich nennt.
- Sowohl in diesem neuen Interface als auch in der zugehörigen Klasse müssen die entfernt aufrufbaren Methoden zusätzliche mit einer `RemoteException` deklariert werden. Entsprechend müssen alle Stellen im Code überarbeitet werden, die die Methode aufrufen, um die potentiell geworfene neue Ausnahmebedingung abzufangen.
  Betrachten wir beispielsweise die Methode `foo`, die entfernt aufgerufen werden soll. Der Programmierer muß nun jeden Aufruf von `foo` in eine `try`-Anweisung einschließen.

  ```
  try {
    server.foo(...);
  } catch (RemoteException e) {
    ... what to do?
  }
  ```

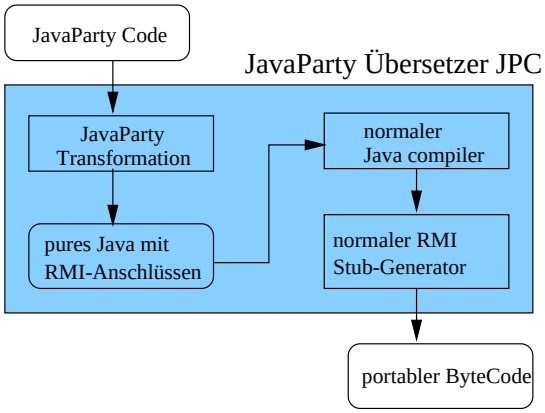
  Während es sinnvoll ist, über WANs betriebene Client-Server-Applikationen gegen Netzwerkfehler abzusichern, ist dies im Fall von eng gekoppelten Clustern von Arbeitsplatzrechnern nicht erforderlich.
  Ein weiterer Nachteil der zusätzlichen Ausnahmebedingung ist, daß im JDK enthaltene Standard-Interfaces nicht mehr genutzt werden können. So kann z.B. `java.lang.Enumeration` nicht mehr verwendet werden, was unter Umständen erheblichen Einfluß auf den gegebenen Code hat.
- In der Regel schreibt RMI vor, daß entfernte Klassen `UnicastRemoteObject` erweitern. Da Java allerdings keine Mehrfachvererbung unterstützt, kann diese Vorschrift zu einer vollständigen Umorganisation des gegebenen Codes führen.[3]

[3] Statt `UnicastRemoteObject` zu erweitern kann der Programmierer explizit Code kopieren, der ansonsten geerbt wird. Dieses Vorgehen legt aber noch mehr Details von RMI offen als das ohnehin der Fall ist.

Abschnitt 4 beschreibt im Detail, wie JavaParty transparente Methodenaufrufe realisiert.

4 Entwurf und Implementierung

Wir haben JavaParty in Form einer Vorverarbeitungsphase in unseren Java-Übersetzer EspressoGrinder [19] eingebaut, wie es folgendes Diagramm darstellt.

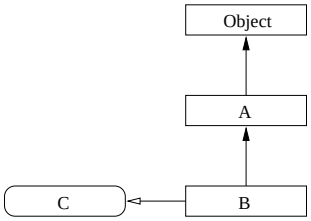


Gegenwärtig nutzt JavaParty RMI als Zielkommunikationssystem und erbt entsprechend dessen Vorteile, z.B. das Vorhandensein eines verteilten Garbage-Collectors. Der JavaParty-Übersetzer kann sowohl unmittelbar ByteCode als auch gewöhnliches Java plus RMI-Konstrukte erzeugen, das dann mit jedem Standard-Java-Übersetzer, z.B. `javac`, in ByteCode überführt werden kann. Die von RMI benötigten Stubs werden durch den von Sun bereitgestellten RMI-Übersetzer `rmic` erzeugt.

In den folgenden Abschnitten stellen wir die Transformation von JavaParty nach Java und RMI im Einzelnen dar. Aufgrund der geforderten Kürze müssen viele Dateils (z.B. Gleichheit von Objekten, `this`, Synchronisation etc.) dabei unberücksichtigt bleiben. Die Transformation wird an folgendem Beispiel durchgeführt:

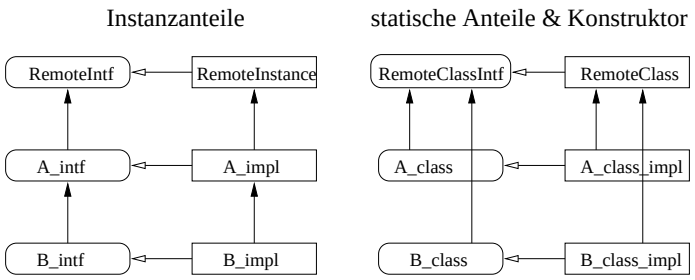
```
remote class B extends A implements C {
  T x = I;                  // instance variable
  static U y = J;           // static variable
  T foo(V z) { P }          // method
  static void foo2() { Q }  // static method
  static { R }              // static block
  B(T z) { S }              // constructor
}
```

Graphisch dargestellt sieht die Klassenhierarchie wie folgt aus. Pfeile mit gefüllten Spitzen drücken eine Unterklassenbeziehung (`extends`) aus, während Pfeile mit leeren Spitzen die Implementation (`implements`) eines Interfaces repräsentieren. Klassen werden durch Rechtecke, Interfaces durch Ovale modelliert.



Instanzanteile und statische Anteile. Die Bestandteile einer Klassendeklaration können in solche eingeteilt werden, die statisch, also für alle Objekte dieser Klasse gemeinsam sind, und solche, die für jede Instanz dieser Klasse eigene Werte haben (Instanzmethoden bzw. Instanzvariablen). RMI kann lediglich Instanzmethoden von entfernten Objekten handhaben. Alle anderen Klassenbestandteile müssen also so umgeformt werden, daß sie dennoch mit Hilfe von RMI genutzt werden können.

Die Idee besteht darin, für die statischen Anteile eine neue Klasse einzuführen, von der genau ein einziges Objekt instantiiert wird. Die Instanzvariablen und Instanzmethoden dieses Objektes übernehmen die Funktion der statischen Klassenbestandteile der gegebenen Klasse und können mit Hilfe von RMI angesprochen werden. Aus der gegebenen Klasse `B` werden also zwei Klassen erzeugt, `B_impl` für die Instanzbestandteile und `B_class_impl` für die statischen Anteile. Damit beide Klassen mit Hilfe von RMI verwendet werden können, müssen zusätzlich noch Interfaces deklariert werden, die alle entfernt ausführbaren Methoden anführen. Zusätzlich zu den `*_impl`-Klassen werden die Interfaces `B_intf` und `B_class` benötigt. Das folgende, noch unvollständige Diagramm stellt die Situation im Zusammenhang dar. In der obersten Schicht finden sich Klassen und Interfaces, die vom JavaParty-System vorgehalten werden und die geeignete RMI-Klassen erweitern.



Für die statischen Bestandteile von `B` wird der folgende Code erzeugt:

```
interface B_class extends RemoteClassIntf {...}

class B_class_impl extends RemoteClass
implements B_class {
  U y;
  // Initialization of the class object
```

```
  protected void _init() {
    y = J;       // static variables
    toRemote(R); // static block
  }
  public void foo2() throws RemoteException {
    toRemote(Q)
  }
  // constructor for instance part
  public B_intf _new(T z) throws RemoteException {
    return new B_impl(z);
  }
}
```

Wie von RMI gefordert, deklariert der erzeugte Code das mögliche Auftreten von `RemoteExceptions`. Für jede dynamisch geladene Klasse wird auf einem Knoten ein Objekt der Klasse `B_class_impl` erzeugt, wobei die von JavaParty realisierte und vom Programmierer modifizierbare Verteilungsstrategie den Knoten auswählt. Zur Initialisierung wird vom Laufzeitsystem die Methode `_init()` aufgerufen. In diesem (und in den folgenden) Code-Fragmenten benutzen wir die Kurzform `toRemote`, um anzuzeigen, daß die dargestellte Transformation noch unvollständig ist.

Die letzte Methode, `_new(T z)`, erfordert weitere Erläuterungen. In JavaParty ist es möglich, Objekte auf entfernten Knoten zu erzeugen. Es gibt zwei unterschiedliche Wege, diese Funktionalität zu realisieren, obwohl RMI dies selbst nicht vorsieht. Der naheliegende Ansatz ist es, das Objekt lokal zu erzeugen und es dann auf den Zielknoten umzuziehen. Da es jedoch wesentlich schneller ist, das Objekt direkt auf dem Zielknoten zu erzeugen, muß ein Weg gefunden werden, den Objekt-Konstruktor auf der Zielmaschine aufzurufen. Dazu benötigt man auf der Zielmaschine einen Agenten, der dies im Auftrag ausführt. Genau dazu ist die Methode `_new` erforderlich: Auf jedem Knoten ist ein Agent vom Typ `B_class_impl` instantiiert, dessen Methode `_new` entfernt aufgerufen werden kann und als Ergebnis eine entfernte Referenz auf ein neues Objekt vom Typ `B_impl` zurückliefert.

Insgesamt wird also auf jedem Knoten ein Objekt vom Typ `B_class_impl` benötigt, wobei allerdings nur eines dieser Objekte die statischen Klassenbestandteile von `B` realisiert.[4]

Die Instanzanteile der Klasse `B` werden ebenfalls in ein Interface und eine Klasse überführt. (Die Implementation von `B_impl` ist zur Vereinfachung der Erläuterung unvollständig; sie wird später komplettiert.)

```
interface B_intf extends A_intf {...}

class B_impl extends A_impl implements B_intf {
  T v = I; // instance variable
  public T foo(V z) // instance method
  throws RemoteException {
    toRemote(P)
  }
```

[4] Aufgrund der Kombination von Erzeugungsagenten und statischen Klassenbestandteilen ist in Zukunft eine Replikation der statischen Anteile leicht zu ergänzen.

```
  // constructor
  B_impl(T z) throws RemoteException {
    toRemote(S)
  }
  public final T _get_B_v() throws RemoteException { //access method
    return v;
  }
  public final T _set_B_v(T _x) throws RemoteException { //access method
    return v = _x;
  }
  // if type T is a numeric base type:
  // access method
  public final T _inc_B_v(T _x, boolean _postfix)
  throws RemoteException {
    T _e = v;
    v += _x;
    if (_postfix) return _e; else return v;
  }
}
```

In obigem Code lassen sich eine Variable `v`, eine Methode `foo` und ein Konstruktor `B_impl` erkennen. Drei zusätzliche Methoden implementieren verschiedene Zugriffe auf die Variable. Nicht dargestellt ist die Vielzahl der Zugriffsmethoden, die für Array-Variablen generiert werden. Bei der Konstruktion der Methodennamen müssen Paket- und Klassennamen berücksichtigt werden, um den Unterschied zu verbergen, den Java bei der Auflösung von Methoden- und Variablennamen vornimmt. Wie von RMI gefordert, deklarieren alle Methoden das potentielle Auftreten einer `RemoteException`.

Handles, Lokalität und Objektmigration. Der gründliche Leser wird bemerkt haben, daß die entfernten Methoden noch immer `RemoteExceptions` auslösen. Um den aufrufenden Code nicht auch noch transformieren zu müssen, führen wir zusätzliche Objekte ein, die wir als „Handles“ bezeichnen. Um die getrennte Übersetzbarkeit der JavaParty-Klassen zu gewährleisten, erhält dieses Handle den ursprünglichen Namen der `remote`-Klasse.

```
class B extends A implements C {
  ...
  T foo(V z) {          // instance method
    while (true)
      try { return ((B_intf)ref).foo(z); }
      catch (MovedException _e)
          {_adaptRef(_e);}
      catch (RemoteException _e)
          {_handleRemoteException("B.foo",_e);}
  }
  static void foo2() {  // static method
    try { ((B_class)RuntimeEnvironment.
          getClassObj("B")).foo2(); }
    catch (RemoteException _e)
          {_handleRemoteException("B.foo2", _e);}
  }
  ...
}
```

Die Vererbungsbeziehung zwischen den Handle-Klassen entspricht derjenigen zwischen den urspünglichen `remote`-Klassen. Für alle Methoden der ursprünglichen Klasse befinden sich Methoden von identischer Signatur in der Handle-Klasse, die *keine* `RemoteException` auslösen. Das Handle hat ferner die Aufgabe, ankommende Methodenaufrufe entweder an das Klassenobjekt umzuleiten, das den statischen Anteil der Klasse implementiert, oder an das Instanzobjekt, das auf irgendeinem der Knoten realisiert ist. Dazu wird die Handle-interne Referenz `ref` verwendet.

Dieser Zeiger `ref` ist darüberhinaus der Schlüssel zur Objektmigration in JavaParty. Wenn ein entferntes Objekt von einem Knoten zu einem anderen umzieht, dann hinterläßt es am ursprünglichen Ort einen Stellvertreter. Wenn bei diesem Stellvertreter ein Methodenaufruf ankommt, dann löst der Stellvertreter beim entfernten Aufrufer eine `MovedException` aus und übermittelt gleichzeitig die neue Adresse des Objekts. Das Handle reagiert auf diese Ausnahmebedingung, korrigiert den `ref`-Zeiger und ruft die Methode erneut, diesmal aber an der neuen Adresse des entfernten Objekts auf. Das erklärt die `while`-Scheife in obigem Code-Fragment.

Bevor ein Objekt (mit Hilfe der Serialisierung) umziehen kann, muß sichergestellt werden, daß gerade keine seiner Methoden ausgeführt wird. Die Transformation von `B_impl` muß also noch verfeinert werden. Vor dem eigentlichen Methodenrumpf wird `_enter()` aufgerufen, anschließend `_leave()`. Damit `_leave()` in jedem Fall ausgeführt wird, wird der Methodenrumpf in eine `try`-Anweisung eingeschlossen und `_leave()` im `finally`-Block plaziert. Der folgende Code zeigt die notwendigen Ergänzungen exemplarisch für `foo`; die anderen Methoden von `B_impl` sind entsprechend zu erweitern.

```
class B_impl extends A_impl implements B_intf {
  ...
  public T foo(V z)
  throws MovedException,
  throws RemoteException {
    _enter();
    try { toRemote(P) }
    finally { _leave(); }
  }
  ...
}
```

5 Benchmark-Resultate

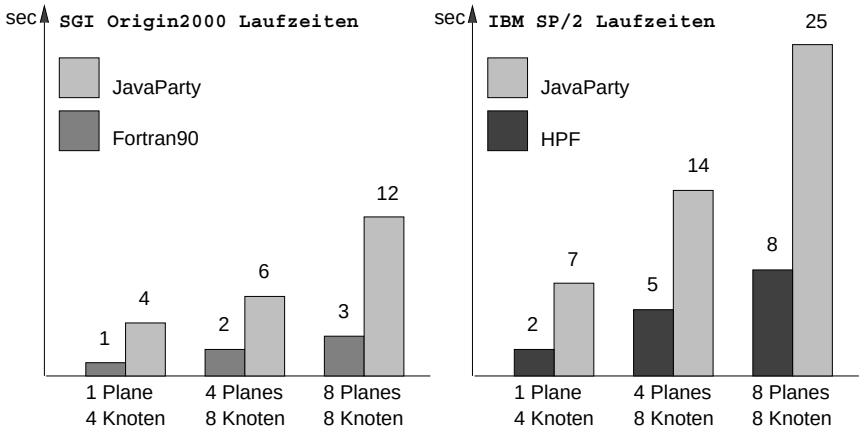
Wir haben datenintensive geophysikalische Basisalgorithmen, nämlich die Geschwindigkeitsanalyse Veltran und die Kirchhoff-Migration, implementiert, um die Leistungsfähigkeit von JavaParty zu untersuchen. Diese Basisalgorithmen werden in der Geophysik verwendet, um Strukturen der inneren Erdschichten mit Hilfe von Schallwellenreflektionen zu entschlüsseln. Da zur Abdeckung eines interessanten Gebiets Sensordaten im Umfang von Terra-Bytes anfallen können, ist die Leistung dieser und anderer geophysikalischer Algorithmen entscheidend. Die

geophysikalischen Zusammenhänge und die Details dieses Benchmarks können in [11] nachgelesen werden.

In Zusammenarbeit mit dem Stanford Exploration Project [3] haben wir diese Algorithmen in JavaParty, in HPF und in Fortran90 implementiert und die Laufzeiten erstens auf einem IBM SP/2 Parallelrechner mit verteiltem Speicher und 8 Knoten und zweitens auf einem SGI Origin 2000 Parallelrechner mit gemeinsamem Speicher vermessen.

Die JavaParty-Implementierung verwendet auf der IBM SP/2 das JDK 1.1.4. Auf jedem Knoten ist eine separate JVM gestartet, die untereinander mit Hilfe von RMI kommunizieren. Für die HPF-Messungen wurde der Übersetzer der Portland Group (Version 2.2.) benutzt. Auf der SGI Origin 2000 verwendeten wir die beta-Version des JDK 1.1.5, die native Threads unterstützt, und den Fortran-Übersetzer von SGI.

Während auf beiden Maschinen hochoptimierende Fortran-Übersetzer verwendet wurden, beruht die Leistung von Java nicht auf nativem Code, sondern lediglich auf just-in-time-Übersetzern.



Auf der SGI ist die JavaParty-Version um den Faktor 4 langsamer als das äquivalente Fortran90-Programm. Auf der SP/2 beobachten wir einen Verlangsamungsfaktor von 3. Diese Verlangsamungen, die aufgrund der verkürzten Programmentwicklungszeiten auf Akzeptanz stoßen, sind vermutlich zu einem erheblichen Anteil auf die impliziten Array-Bereichsüberprüfungen zurückzuführen, die Java bei jedem einzelnen Array-Zugriff durchführt.

Während das JavaParty-Programm sich automatisch an die Größe der Eingabedaten (gemessen in Anzahl der „Planes“, ein genaueres Verständnis ist an dieser Stelle nicht erforderlich) und die zur Verfügung stehende Knotenzahl anpaßt, fehlt den Fortran-Programmen diese Flexibilität. Für jede einzelne Messung mußten einige Konstanten im Programmtext verändert und das Programm neu übersetzt werden. Ohne diese manuellen Änderungen und Nachübersetzungen wäre die jeweils langsamste Laufzeit überall zu beobachten gewesen. Beispielsweise hätte es dann 8 Sekunden gedauert, um auf der SP/2 mit 4 Knoten eine „Plane“ zu bearbeiten.

Aus zwei Gründen erwarten wir für Java in der nahen Zukunft erhebliche Leistungsverbesserungen. Erstens waren wir gezwungen, das JDK 1.1.x für die Messungen zu verwenden, da neuere Versionen noch nicht portiert waren. Spätere Versionen haben aber auf anderen Plattformen erhebliche Leistungsverbesserungen gebracht, so daß ähnliche Verbesserungen auch hier zu erwarten sind. Insbesondere verbesserten sich RMI, die Unterstützung nativer Threads und die just-in-time-Übersetzungstechnik (HotSpot). Zweitens sind statische Java-Übersetzer in ersten Versionen verfügbar, z.B. der High Performance Java Compiler von IBM [10], der durch Verwendung klassischer Optimierungstechniken erhebliche Leistungsverbesserungen erwarten läßt.

6 Verwandte Arbeiten

Parallele objektorientierte Sprachen. Von den über hundert in [20] untersuchten Sprachen betrachten mehr als die Hälfte die Probleme der Objektverteilung und -lokalität überhaupt nicht. Dafür gibt es unterschiedliche Gründe. Einige Sprachen sind lediglich prototypisch und für einen einzelnen Arbeitsplatzrechner realisiert worden, auf dem naturgemäß Netzwerklatenzzeiten keine Rolle spielen. Die Entwickler sind vorwiegend daran interessiert, Sprach- und Synchronisationsmittel in objektorientiertem Zusammenhang grundsätzlich zu untersuchen. Aus Sicht dieser Forscher sind Threads und explizite Synchronisierung, wie sie in Java vorkommen, nicht optimal, da sie zu unterschiedlichsten Formen der „Inheritance Anomaly“ [17] führen können.[5] Andere Sprachen beschränken sich von vorneherein auf Parallelrechner mit gemeinsamem Speicher und verlassen sich auf das jeweils vorhandene Cache-System.

JavaParty hat einige Vorteile, die andere parallele objektorientierte Programmiersprachen nicht haben. JavaParty ist fast mit Java identisch, wodurch es ohne weiteren Lernaufwand von vielen Java-Programmierern verwendet werden kann. Im Gegensatz zu anderen vorgeschlagenen parallelen objektorientierten Programmiersprachen ist JavaParty entwurfsbedingt auf allen Plattformen in gleicher Weise verfügbar und profitiert uneingeschränkt von Leistungssteigerungen, die der Marktdruck von Java-Basistechnologie verlangt.

Parallelität in Java. Obwohl Thread-basierte Parallelität in Java vorhanden ist, wird dies von einigen Forschern als nicht ausreichend bewertet. Einige Gruppen haben daher Sprachmittel für Datenparallelismus zu Java hinzugefügt [9];

[5] Eine Minimaleinführung in „Inheritance Anomaly“: Das Problem besteht darin, daß die Code-Zeilen, die die Synchronisationserfordernisse ausdrücken, über alle Methoden einer Klasse verteilt sind. Wenn nun eine Unterklasse leicht abweichende Anforderungen an die Synchronisation hat, dann tritt die Anomalie in Erscheinung: Anstatt fast alle Methoden der Oberklasse erben zu können, müssen fast alle Methoden der Oberklasse textuell kopiert und dann manuell hinsichtlich der geänderten Synchronisationsanforderungen verändert werden. Dabei bleibt in der Regel der in den Methoden implementierte Algorithmus unverändert, nur die Synchronisation ist modifiziert. Code-Duplikation erschwert Code-Wartung und ist daher zu vermeiden.

es existiert auch eine Erweiterung von JavaParty, die eine `forall`-Anweisung ergänzt [21]. Andere Gruppen ergänzten Ausdrucksmittel für andere Paradigmen [13, 15]. Während die meisten dieser Systeme maschinenabhängige Bibliotheken oder nicht-portable Implementierungen der JVM voraussetzen, bleibt JavaParty so nahe an Java wie möglich und funktioniert – auch wegen der Möglichkeit, Java-Quellcode zu erzeugen – in jeder standardkonformen Java-Umgebung.

Objektmigration. Die positiven Effekte von Objektmigration sind beispielsweise im Emerald-Projekt dokumentiert [12]. Die Integration der Migration in Laufzeit- und Übersetzungszeitoptimierungen ist Gegenstand laufender Forschungsarbeiten in der JavaParty-Gruppe.

Zielplattform. Derzeit basiert die Transformation von JavaParty auf RMI, einem Teil der JDK-Distribution. Daher sind JavaParty-Programme auf allen wichtigen Systemplattformen, auch in heterogenen Umgebungen, ablauffähig.

Es gibt dazu allerdings Alternativen: CORBA unterstützt zwar eine Vielzahl von Sprachen und ist weniger abhängig von Java, jedoch fehlen wichtige RMI-Eigenschaften von denen wir in der derzeitigen Implementierung Gebrauch machen, z.B. der integrierte verteilte Garbage-Collector.

"Horb" [8] ist mit RMI dahingehend vergleichbar, daß verteilte Methodenaufrufe ermöglicht werden. Horb hätte daher, insbesondere weil der Autor einen deutlichen Geschwindigkeitsvorteil gegenüber RMI postuliert, als Zielsystem der JavaParty-Transformation dienen können. Wir haben uns aber dennoch für RMI entschieden, weil RMI erstens Teil der Standard-JDK-Distribution ist und weil zweitens zu hoffen ist, daß RMI wegen des entstehenden Marktdrucks auf lange Sicht Horb überholen wird.

Derzeit verfolgen wir im Rahmen des JavaParty-Projekts zwei Studien. Einerseits versuchen wir eine optimierte, aber kompatible Implementierung von RMI und der Serialisierung zu erstellen, die bessere Laufzeiten aufweist. Andererseits untersuchen wir die Möglichkeit, direkt auf Sockets oder UDP-Pakete abzubilden und damit den RMI-Overhead vollständig zu umgehen.

Andere entfernte Java-Objekte. Uns sind zwei andere Systeme bekannt, die Java um transparente entfernte Objekte erweitern.

Im Gegensatz zu JavaParty führt „Remote Objects in Java" (ROJ) [18] ein neues Schlüsselwort `remotenew` ein, das verwendet werden muß, um ein Objekt, das später nicht mehr migrieren kann, auf einem bestimmten Knoten anzulegen. Der Programmierer muß also durch gezielte manuelle Objektplazierung Lokalität herstellen. Da das neue Schlüsselwort auf einen Operationscode abgebildet wird, um den der Befehlsvorrat des ByteCodes erweitert wurde, wird eine veränderte JVM benötigt. Wir sind der Meinung, daß das ein schwerer Nachteil ist, weil ROJ dadurch weder auf allen Plattformen verfügbar sein kann noch leicht von Fortschritten der just-in-time-Übersetzung profitieren kann.

Eine Einschränkung von ROJ ist, daß nur primitive Typen als Parameter von entfernten Methodenaufrufen verwendet werden dürfen. Eine solche Ein-

schränkung hätte die Portierung unserer Benchmark-Programme nach JavaParty erheblich erschwert.

Interessant ist, daß ROJ ohne gemeinsames Dateisystem auskommt; stattdessen wird ByteCode über das Netzwerk verschickt.

"Java/DSM" [26] implementiert Java auf der Basis von Treadmarks [14], einem in Software auf einem DMP realisierten gemeinsamen Speicher. Wie ROJ benötigt auch Java/DSM eine Spezialimplementierung der JVM. Ferner müssen in diesem System spezielle Vorkehrungen für heterogene Umgebungen getroffen werden, die durch die Verwendung von RMI bereits gelöst sind. Schließlich hoffen wir, daß JavaParty durch informierte Lokalitätsentscheidungen bessere Laufzeiten erreicht, als dies Java/DSM durch einen allgemeinen und auf Seitengrößen basierenden Cache-Ansatz möglich ist.

7 Zusammenfassung

JavaParty ermöglicht die Programmierung von DMPs und von Clustern von Arbeitsrechnern in Java. Dieselben Programme sind auch auf SMPs ausführbar. Sie sind wesentlich kürzer als handgeschriebene Programme, die Sockets oder RMI verwenden, und passen sich dynamisch der zugrundeliegenden Netzkonfiguration an. Mit JavaParty können Laufzeiten erreicht werden, die für geophysikalische Anwendungen mit äquivalenten Fortran-Implementierungen konkurrenzfähig sind.

Das JavaParty-System, bestehend aus einem Laufzeitsystem und einem vollständigen Übersetzer, der neben ByteCode auch puren Java-Code erzeugen kann, ist für nicht-kommerzielle Verwendung frei erhältlich. Weitere Informationen finden sich unter `http://wwwipd.ira.uka.de/JavaParty/`

Danksagung. Wir möchten an dieser Stelle allen Mitgliedern der JavaParty-Gruppe danken. Sun Microsystems Deutschland gab uns die Gelegenheit, JavaParty auf der CeBIT'98 auszustellen. Sowohl das Maui High Performance Computing Center als auch das Rechenzentrum der Universität Karlsruhe ermöglichten dankenswerterweise den Zugriff auf die SP/2 und die SGI Origin2000.

Literatur

1. T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.
2. N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
3. J. Clearbout and B. Biondi. Geophysics in object-oriented numerics (GOON): Informal conference. In *Stanford Exploration Project Report No. 93.* October 1996. `http://sepwww.stanford.edu/sep`
4. J.T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Elsevier Science Publishers, Holland, 1992.

5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
7. J. Gosling, F. Yellin, and The Java Team. *The Java Application Programming Interface*, volume 1 – Core Packages. Addison-Wesley, 1996.
8. S. Hirano. Horb: net computing, `http://ring.etl.go.jp/openlab/horb/`, 1996.
9. S. Flynn Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, June 1997.
10. IBM. High performance compiler for Java. `http://www.alphaWorks.ibm.com`
11. M. Jacob. Implementing Large-Scale Geophysical Algorithms with Java: A Feasibility Study. Master's thesis, University of Karlsruhe, Dept. of Informatics, 1997.
12. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
13. L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel Java with global object space. In *Proc. of Conf. on Distributed Processing Technology and Applications*, Las Vegas, Nevada, 1997.
14. P. Keleher, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter Usenix Conf.*, pages 115–131, January 1994.
15. P. Launay and J.-L. Pazat. Integration of control and data parallelism in an object oriented language. In *Proc. of 6th Workshop on Compilers for Parallel Computers (CPC'96)*, Aachen, Germany, December 11–13, 1996.
16. D. Lea. *Concurrent Programming in Java – Design Principles and Patterns*. Addison-Wesley, 1996.
17. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
18. N. Nagaratnam and A. Srinivasan. Remote objects in Java. In *IASTED Intl. Conf. on Networks*, January 1996.
19. M. Odersky, M. Philippsen. Espresso, `http://wwwipd.ira.uka.de/~espresso`, 1996.
20. M. Philippsen. Imperative concurrent object-oriented languages. Technical Report 95-050, International Computer Science Institute, Berkeley, August 1995.
21. M. Philippsen. Data parallelism in Java. In J. Schaefer, editor, *High Performance Computing Systems and Applications*. Kluwer Academic Publishers, 1998. to appear.
22. M. Philippsen, M. Jacob, and M. Karrenbach. Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java. *Informatik—Forschung und Entwicklung*, 13(2):72–78, 1998.
23. DFG Forschergruppe RESH – Rechnernetze als Basis für Supercomputer und Hochleistungsdatenbanken. `http://wwwipd.ira.uka.de/RESH`
24. Sun Microsystems Inc., Mountain View, CA. *Java Remote Method Invocation Specification, beta draft*, 1996.
25. T.M. Warschko, J.M. Blum, and W.F. Tichy. The ParaStation Project: Using Workstations as Building Blocks for Parallel Computing. In *Intl. Conf. on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96)*, pages 375–386, Sunnyvale, CA, August 9–11, 1996.
26. W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.