Parallele und verteilte Programmierung mittels Pthreads und Rthreads

Bernd Dreier und Markus Zahn
Universität Augsburg
Institut für Informatik
D-86135 Augsburg
{dreier,zahn}@informatik.uni-augsburg.de

Theo Ungerer
Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
D-76128 Karlsruhe
ungerer@informatik.uni-karlsruhe.de

November 1997

Das Ziel des Rthreads-Projekts (Remote Threads) ist es, parallele Anwendungen für Rechnernetze verfügbar zu machen. Insbesondere vielfädige (multithreaded) Programme, die für Multiprozessor-Workstations geschrieben wurden, sollen ohne Neuprogrammierung auf ein Rechnernetz transferierbar sein. Um dies zu erreichen, haben wir das DSM-System Rthreads entwickelt, welches zur Beschreibung der Parallelität und der Synchronisation dieselben Primitive wie POSIX Threads (Pthreads) sowie zusätzliche Funktionen für den Zugriff auf den gemeinsam genutzten Speicher verwendet. Die Rthreads-Funktionsbibliotheken setzen auf einem der verteilten Systeme DCE, PVM, MPI oder Active Messages auf. Unser Ansatz erlaubt es, POSIX 1003.4a-konforme vielfädige Programme automatisch in verteilte Anwendungen zu transformieren. Die Übersetzung von Pthread-Programmen in Rthread-Programme wird von einem Präcompiler ausgeführt. Nach der Präcompilerphase kann der Programmierer eine Optimierung auf Quellcodeebene durchführen.

1 Einleitung

In den letzten Jahren hat das verteilte Rechnen, also der Zusammenschluß mehrerer Computer zur Lösung eines Problems, immer mehr Verbreitung gefunden. Insbesondere vernetzte Standard-Workstations, sogenannte Workstation-Cluster, werden immer häufiger als virtueller Parallelrechner verwendet. Heute werden dabei in der Praxis vorwiegend Message-Passing-Softwarepakete wie PVM oder MPI verwendet. Bei der Message-Passing-(MP-)Programmierung wird die gesamte Kommunikation und Synchronisation der teilnehmenden Knoten über das Versenden von Nachrichten vorgenommen. Die korrekte Berücksichtigung der Datenabhängigkeiten zwischen den einzelnen Knoten durch den Nachrichtenaustausch wird für komplexe Anwendungen sehr schnell aufwendig, da der Programmierer nur schwerlich den Überblick behalten kann. Bei der parallelen Programmierung mit gemeinsamem Speicher (shared memory), wie sie z.B. von symmetrischen Multiprozessoren bekannt ist, arbeiten die teilnehmenden Knoten auf gemeinsamen globalen Daten und synchronisieren sich auch über diese. Dieses Programmier-

paradigma wird insbesondere für kompliziertere Algorithmen als einfacher als die MP-Programmierung empfunden, da es mehr der bekannten, sequentiellen Programmierung ähnelt.

Unabhängige Knoten eines Netzwerks, wie die Workstations in einem Cluster, besitzen jedoch keinen gemeinsamen Speicher. Um das einfacher zu beherrschende Shared-Memory-Programmierparadigma auch für das verteilte Rechnen zugänglich zu machen, wurden Ende der achtziger Jahre die ersten „Distributed-Shared-Memory (DSM)"-Systeme für Workstation-Cluster entwickelt. Dabei wird den einzelnen Knoten der physikalisch nicht vorhandene gemeinsame Speicher durch Hard- und/oder Softwaremethoden zur Verfügung gestellt.

Da der gemeinsame Speicher bei DSM-Systemen nur virtuell vorhanden ist, müssen bei Schreib- oder Leseanfragen zusätzliche Mechanismen in ein verteiltes Programm eingebunden werden. Die ersten DSM-Systeme verwendeten hierzu die virtuellen Speichermechanismen der zugrundeliegenden Betriebssysteme zur Verwaltung der über Maschinengrenzen hinweg gemeinsamen benutzten Speicherseiten. Später wurde der für Schreib-/Leseanfragen zusätzlich notwendige Code auch durch modifizierte Compiler eingefügt. Die einfachsten Systeme verwalten die gemeinsamen Daten indem sämtliche Speicherzugriffe zu einem oder mehreren zuständigen Managern dirigiert werden. Um den hohen Kommunikationsaufwand zu reduzieren, wurden lokale Puffer eingeführt, die bei jeder Änderung des Speichers durch einen der teilnehmenden Knoten aktualisiert oder invalidiert werden. Trotzdem ist der Kommunikationsaufwand im Vergleich zu Message-Passing-Systemen sehr viel höher. Bei der MP-Programmierung muß der Programmierer selbst dafür Sorge tragen, daß jeder Knoten die benötigten Daten zum richtigen Zeitpunkt zur Verfügung hat. Da ein DSM-System die Bedürfnisse des jeweiligen Algorithmus nicht kennt, muß es bei der Aktualisierung der Daten auf den verschiedenen Knoten eine defensive Strategie einschlagen. Solche Strategien wurden in den letzten Jahren in der Form von sogenannten Konsistenzmodellen entwickelt. Das Konsistenzmodell eines DSM-Systems beschreibt, wann die Daten eines Knotens relativ zu anderen Speicherzugriffen konsistent, also auf dem aktuellen Stand, sind. Durch die Entwicklung schwächerer Konsistenzmodelle, bei denen die Daten nur noch relativ zu Synchronisationsoperationen aktualisiert werden, konnte der im Vergleich zu MP-Systemen erhöhte Kommunikationsaufwand bereits erheblich reduziert werden. Das bezüglich der Performance vielversprechendste Konsistenzmodell ist derzeit die *release*-Konsistenz [GL+90] in Verbindung mit einem *lazy*-Protokoll [KCZ92]. Bei sogenannten *lazy*-Protokollen werden die gemeinsamen Daten nicht bereits aktualisiert, wenn eine Inkonsistenz entsteht, sondern erst dann, wenn sie bemerkt werden kann. In der Praxis bedeutet dies, daß die Daten erst dann aktualisiert werden, wenn sie auch tatsächlich benötigt werden — es werden also niemals „zuviel" Daten übertragen. Der einzige Overhead in der Nachrichtenmenge, der hierbei noch entsteht, liegt in den Protokollinformationen, die benötigt werden, um festzustellen, welche Daten inkonsistent sind. Da diese Protokollinformationen ausschließlich bei sowieso notwendigen Synchronisationsaufrufen mit übertragen werden, ist auch eine Verringerung der Nachrichtenanzahl durch die Entwicklung anderer Konsistenzmodelle kaum zu erwarten.

Das niedrige Kommunikationsbedürfnis von MP-Systemen kann hierbei ohnehin nicht ganz erreicht werden, da durch die von der reinen Datenübertragung entkoppelten Synchronisationsoperationen des Shared-Memory-Paradigmas zusätzliche Speicherzugriffe notwendig sind.

Die Möglichkeit einer weiteren, entscheidenden Reduzierung der Kommunikation zur Aktualisierung der Daten durch die Weiterentwicklung moderner Konsistenzmodelle, wie z.B. der *lazy-release*-Konsistenz, ist also zumindest fraglich. Unserer Meinung nach liegt der Grund hierfür darin, daß jedes solche Konsistenzmodell einem statischen Schema entspricht, das nicht an den jeweiligen Algorithmus angepaßt werden kann. Eine entscheidende Verbesserung kann unseres Erachtens nur durch eine flexible, dem Algorithmus angepaßte Gestaltung des Konsistenzmodells erfolgen. Durch die Realisierung der Speicherzugriffe über die Speicherverwaltung des Betriebssystems bzw. über modifizierte Compiler bisheriger Systeme wird dem Programmierer jedoch die Möglichkeit genommen, selbst auf die notwendige Kommunikation Einfluß zu nehmen, da die Speicherzugriffe erst zur Lauf- bzw. Übersetzungszeit — nach einem festen, vorgegebenen Schema — eingefügt werden.

Weitere Schwachpunkte bestehender DSM-Systeme sind die fehlende Unterstützung heterogener Netze und die mangelnde Portabilität. Die meisten heute betriebenen Workstation Cluster sind heterogen, meist findet man nicht nur verschiedene Modelle eines Herstellers, sondern auch Workstations verschiedener Hersteller mit unterschiedlichen Betriebssystemen. Es finden sich jedoch kaum DSM-Systeme, die solche Cluster unterstützen. Auch hierbei resultieren die Hauptprobleme aus den bisher üblichen Speicherzugriffsmechanismen, die die Unterschiede bezüglich Typen und Ausrichtung von Daten im Speicher nicht ausreichend beachten. Ein zentraler Grund für den Erfolg der MP-Systeme PVM und später MPI, war ihre hohe Portabilität. So wurde es möglich ein auf einem Workstation Cluster erstelltes und getestetes Programm später ohne Änderungen im Quellcode auf einem Hochleistungsrechner ablaufen zu lassen. Die meisten DSM-Systeme sind auf Grund ihrer Struktur vom Betriebssystem, Compiler oder sogar der zugrundeliegenden Hardwarearchitektur abhängig. Eine Portierung eines solchen Systems (und damit seiner Programme) ist daher extrem aufwendig und wird selten durchgeführt. Threads, die heute in jedem modernen Betriebssystem enthalten sind, werden derzeit von PVM und MPI noch nicht und von DSM-Systemen kaum unterstützt. Damit ist die Nutzung von Multiprozessor-Workstations eines Netzes sowie die Zerlegung eines Problems in feingranulare Teile nicht möglich.

2 Grundkonzepte des Rthreads-Systems

2.1 Überblick über Rthreads

Als Konsequenz der obigen Feststellungen haben wir das DSM-System Rthreads entwickelt. Rthreads verwendet zur Beschreibung der Parallelität und der Synchronisation dieselben Primitive wie POSIX Threads (Pthreads). Aus der Tatsache, daß es mit

Rthreads ermöglicht wird, ein Programm auf mehrere vernetzte (ferne, remote) Knoten zu verteilen und gleichzeitig auf einem Knoten mittels Pthreads parallel zu arbeiten, leitet sich der Name R(emote)Threads ab. Durch die hohe Verfügbarkeit von Pthreads auf heutigen Betriebssystemen und durch ihre Bekanntheit ist eine leichte Portierung von bestehenden Programmen sowie eine einfache Neuentwicklung von Programmen gegeben.

Die Portabilität des Rthreads-Systems selbst wird durch die Implementation mittels Präcompiler und Funktionsbibliotheken erreicht, wobei die Funktionsbibliotheken jeweils auf einem der weitverbreiteten Kommunikationssysteme PVM, MPI, DCE oder Active Messages aufsetzen. Damit ist das Rthreads-System auf allen Parallelrechnern und Workstation-Clustern lauffähig, die eines der oben genannten Pakete und Pthreads unterstützen.

Die aus der erhöhten Kommunikation entstehenden Effizienznachteile von DSM-Systemen versuchen wir durch einen neuen Ansatz bei der Organisation der Speicherzugriffe zu verringern: Ausgehend von einem Rthreads-Programm, das bereits die korrekten Aufrufe zur Beschreibung der Parallelität und Synchronisation enthält, nimmt ein Präcompiler auf Quellcodeebene eine Typanalyse vor und setzt außerdem die entsprechenden Schreib-/Leseaufrufe explizit im Quellcode ein.[1] Die frühere Einschätzung, daß die Behandlung der Schreib-/Leseoperationen durch die Mechanismen der virtuellen Speicherverwaltung einer solchen ausschließlich in Software realisierten Lösung bezüglich der Effizienz generell überlegen ist, wurde von Zekauskas et al. in [ZSB94] widerlegt. Entgegen dem sonst bei existierenden DSM-Systemen üblichen Einfügen der Zugriffe durch den Compiler werden die Speicherzugriffe bei Rthreads auf einer Ebene eingefügt, auf die der Programmierer Zugriff hat. Der Vorteil liegt nun darin, daß der Programmierer später (falls die Effizienz dies erfordert) die Speicherzugriffe flexibel an die Bedürfnisse des Algorithmus' anpassen kann. Damit kann der Kommunikationsaufwand auf das geringstmögliche Maß reduziert werden. Um die Performance des ursprünglich vom Präcompiler erhaltenen Programmes zu verbessern, kann zusätzlich eines der schwächeren Konsistenzmodelle implementiert werden.

Die Behandlung der Speicherzugriffe durch den Präcompiler bildet auch eine Grundlage für die Portabilität und die Unterstützung von heterogenen Systemen durch Rthreads: DSM-Systeme, die auf Compileränderungen oder gar auf die virtuelle Speicherverwaltung eines bestimmten Betriebssystems angewiesen sind, können nur so portabel wie der jeweilige Compiler oder das Betriebssystem sein. Der Rthread-Präcompiler wurde mit den Werkzeugen *lex* und *yacc* entwickelt, die auf nahezu allen heute verwendeten Systemen zu finden sind. Die vom Präcompiler durchgeführte Typanalyse wird zur korrekten Datenübertragung in heterogenen Netzen benötigt.

Der Unterschied zwischen dem Quellcode eines Rthread-Programms und dem eines Pthread-Programms liegt, abgesehen von den Speicherzugriffsoperationen, lediglich in den Namen der Typen und Funktionen zur Synchronisation und Parallelitätsbeschreibung. Daher kann die gesamte Transformation eines Pthread-Programms in ein Rthread-

[1] Bisher wurde lediglich die Typanalyse implementiert.

Programm automatisiert werden. Die gesamte Entwicklung eines Programms kann also lokal mit den gewohnten (Debug-)Werkzeugen erfolgen.[2] Die Transformation in ein verteiltes Rthread-Programm erfolgt danach automatisch. Da Pthreads mittlerweile auch als Backend einiger parallelisierender Compiler unterstützt werden (z.B. SUIF), ist sogar die automatische Übersetzung eines sequentiellen Programms in ein Rthread-Programm denkbar. Die einzige Einschränkung, die wir dabei an zu transformierende Pthread-Programme unter anderem wegen der notwendigen Typinformation zur Datenübertragung in heterogenen Netzen stellen müssen, ist das Verbot von verzeigerten Datenstrukturen innerhalb des gemeinsam benutzten Speichers.

2.2 Bildung des virtuell gemeinsamen Speichers

Die meisten DSM-Systeme, deren Speicherzugriffe ebenfalls explizit im Programmtext angegeben werden (z.B. Adsmith [LKL96] oder Phosphorus [DCM95]), verlangen vom Programm zur Laufzeit die nochmalige Deklaration von Daten, die im verteilt gemeinsamen Speicher liegen sollen. Für diese Daten wird dann vom DSM-System intern nochmals ein Zwischenspeicher des entsprechenden Typs angelegt. Jeder Zugriff auf das Datum muß dann letztendlich über diesen Zwischenspeicher erfolgen. Dadurch wird eine künstliche Trennung der Elemente des DSM und der globalen Variablen vorgenommen, da nur diejenigen globalen Variablen in den DSM übernommen werden, die auch zur Laufzeit noch einmal deklariert werden. Zudem ergibt sich über diesen Ansatz meist eine unnatürliche und umständliche Adressierung der Objekte im DSM, wie wir später sehen werden.

Wir haben durch einen neuen Ansatz zur Bildung des verteilt gemeinsamen Speichers versucht, keine Trennung von globalen Daten und den Daten im DSM vorzunehmen. Diesen Ansatz veranschaulicht Abbildung 1. Jede globale Variable ist automatisch Bestandteil des verteilt gemeinsamen Speichers, wie man es im Falle eines physikalisch gemeinsamen Speichers auch erwarten würde. Eine gesonderte Deklaration derjenigen globalen Daten, die „gemeinsam“ sein sollen, ist nicht nötig. Zusätzlicher Zwischenspeicher für die globalen Daten wird ebenfalls nicht angelegt. Der verteilt gemeinsame Speicher wird aus den Speicherbereichen der globalen Variablen der Knotenprogramme gebildet. Der Zugriff auf ein globales Datum (automatisch Teil des verteilt gemeinsamen Speichers) erfolgt auf den Knoten exakt wie im Fall eines physikalisch gemeinsamen Speichers, was im Bild durch die Doppelpfeile zwischen dem Programm und den globalen Variablen gekennzeichnet ist. Aus der Sicht eines Knotenprogramms besteht der DSM aus seinen eigenen globalen Variablen. Da nun aber jeder Knoten zunächst nur lokal auf „seinen“ globalen Variablen arbeitet, muß an bestimmten Stellen des Programms für den Abgleich der verschiedenen Kopien der globalen Variablen gesorgt werden. Dies veranlassen die Knoten durch geeignete Operationen. Im Bild wird dies durch die vom Programm auf einem Knoten ausgelöste Kommunikation zwischen den globalen Variablen (Doppelpfeil) verschiedener Knotenprogramme dargestellt. Hierbei findet kein Datenaustausch

[2] Das Debugging verteilter Programme ist nach wie vor nicht zufriedenstellend gelöst worden. Durch die zunächst lokale Programmentwicklung kann dieses Problem weitgehend umgangen werden.

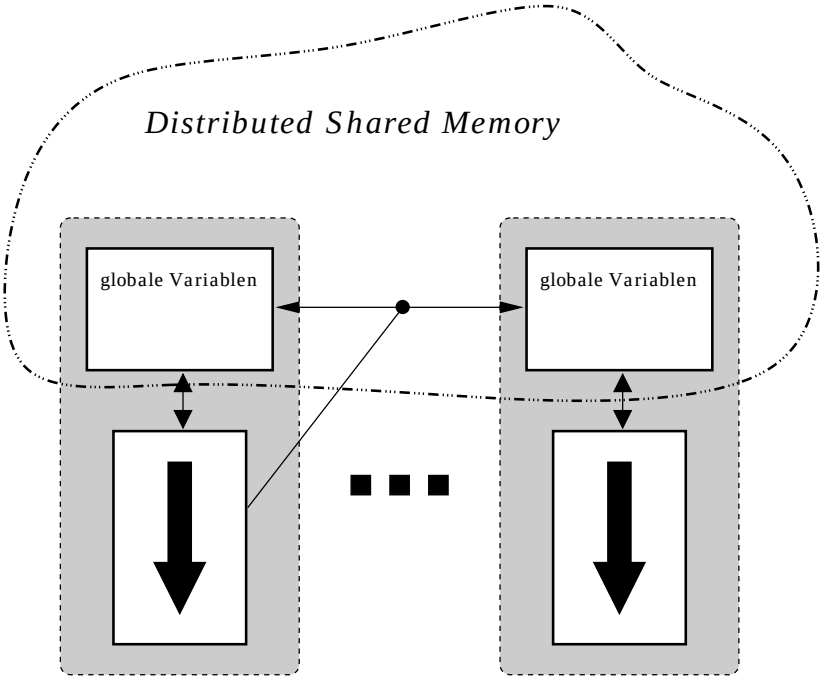

Abbildung 1: Bildung der gemeinsamen Datenobjekte

zwischen dem Programm des Knotens und dem DSM statt. Der Knoten initiiert lediglich einen Abgleich von Objekten des verteilt gemeinsamen Speichers. Jeder Knoten kann also wie gewohnt auf seinen globalen Variablen arbeiten. Jede dieser Variablen stellt im Prinzip eine Kopie eines globalen Datums dar.

Durch diese Architektur beherbergt *jeder* Knoten des Systems automatisch eine Kopie jeder globalen Variablen.[3] Auf dieser Kopie kann er zunächst mit vollen Zugriffsmöglichkeiten lokal arbeiten bevor er den Abgleich mit dem DSM veranlaßt (siehe Abbildung 1).

Die auf verschiedenen Knoten ausgeführten Programme aus Abbildung 1 können durchaus verschieden sein, um z.B. ein Master-Slave Paradigma zu realisieren. Die globalen Variablen sollten jedoch identisch sein, was wohl auch die natürliche Herangehensweise ist.

2.3 Adreßbildung

In einem System mit physikalisch gemeinsamem Speicher werden die (globalen) Daten durch die Angabe ihrer physikalischen Adresse oder auch ihrer Adresse im virtuellen Speicher identifiziert. In einem nicht-seitenbasierten DSM-System wird diese direkte Adressierung über virtuelle Speicheradressen nicht vorgenommen. Die verschiedenen an

[3] Rthreads enthält Mechanismen zur Einsparung von Speicherplatz auf den einzelnen Knoten bei der Verwendung von großen Feldern. Auf diese soll in dieser Arbeit aber nicht näher eingegangen werden.

der Programmausführung beteiligten Knoten verfügen über keinen gemeinsamen Adreßraum und es wird auch kein virtueller Adreßraum im herkömmlichen Sinne über alle Knoten gebildet, wie dies in einem seitenbasierten System geschieht. Daher müssen andere Mechanismen gefunden werden, um ein Datenobjekt des DSM zu adressieren bzw. zu identifizieren.[4]

Bekannte Ansätze hierzu sind einerseits das Einsetzen von eigenen Funktionsaufrufen oder Identifikatoren durch den Compiler, was wiederum Änderungen im Compiler voraussetzt, die wir zugunsten höherer Portabilität vermeiden wollen. Ein anderer Ansatz, der eben die bereits im vorhergehenden Abschnitt erwähnte zusätzliche Deklaration der Daten des DSM verlangt, ist die Identifikation über vom Programmierer bei dieser Deklaration zu vergebende Namen. Bei einem späteren Zugriff kann das jeweilige Datum dann über den als Zeichenkette angegebenen Namen identifiziert werden. Abgesehen von der bereits beschriebenen Trennung von globalen Daten und Objekten im DSM sowie der zusätzlich notwendigen Deklarationsaufrufe führt dieses Verfahren jedoch auch zu Effizienznachteilen: Bei jedem Zugriff auf ein Datum im DSM muß seine identifizierende Zeichenkette in einer Tabelle, die die Namen aller Daten im DSM enthält, gesucht werden. Bei einer großen Zahl von Daten im DSM können diese Vergleiche von Zeichenketten zu erheblichen Leistungseinbußen führen, zumal sie bei *jedem* Zugriff auf ein Datum im DSM ausgeführt werden müssen. Weiterhin führt dieses Verfahren auch etwas von einem möglichst natürlichen Zugriff weg, da der Programmierer eine ihm sinnvoll erscheinende weitere Deklaration der Daten vornehmen muß und über einen im Prinzip beliebig wählbaren, vom eigentlichen globalen Datum unabhängigen Namen auf das Datum zugreift.

In unserem System wird das jeweilige Datum für einen Abgleich der Daten im DSM ebenfalls durch denselben Namen identifiziert, wie er auch sonst im Quelltext vorkommt, eine zusätzliche Deklaration ist nicht notwendig! Für diese Angabe würde normalerweise durch den Compiler die Adresse des Datums auf dem jeweiligen Knoten (welche zur Identifikation eines Datums in einem System ohne gemeinsamen Speicher nicht ausreicht) eingesetzt. Damit das DSM-System mit dieser Form von Aufrufen umgehen kann, muß es also Informationen über die globalen Daten erhalten. Diese Aufgabe übernimmt nun ein Präcompiler. Der Präcompiler wurde mittels der Werkzeuge *lex* und *yacc* in *C* entwickelt und arbeitet auf *C*-Quellcode, sodaß er auf beinahe jedem heute betriebenen System einsatzfähig ist und keine Einschränkungen bezüglich der Portabilität ergibt.

Wie in Abbildung 2 veranschaulicht, analysiert der Präcompiler den Quellcode bezüglich seiner globalen Variablen. Daraus erzeugt er zwei Tabellen: Die erste Tabelle enthält für jede globale Variable eine Marke, die den eigentlichen Identifikator für ein globales Datum darstellt. Der Wert der Marke ist ein Index in die zweite Tabelle, die zu jedem über die Marke identifizierten Datum einen Zeiger auf dieses Datum im Adreßraum des jeweiligen Knotens sowie den Typ des Datums speichert. Dadurch kann die Identifikation der Daten des DSM über diese Marken erfolgen, die über ihren Index in die zweite

[4] Da diese Identifikation nicht mehr über Adressen im eigentlichen Sinne stattfindet, sprechen wir im folgenden auch von Identifikatoren anstatt von Adressen.

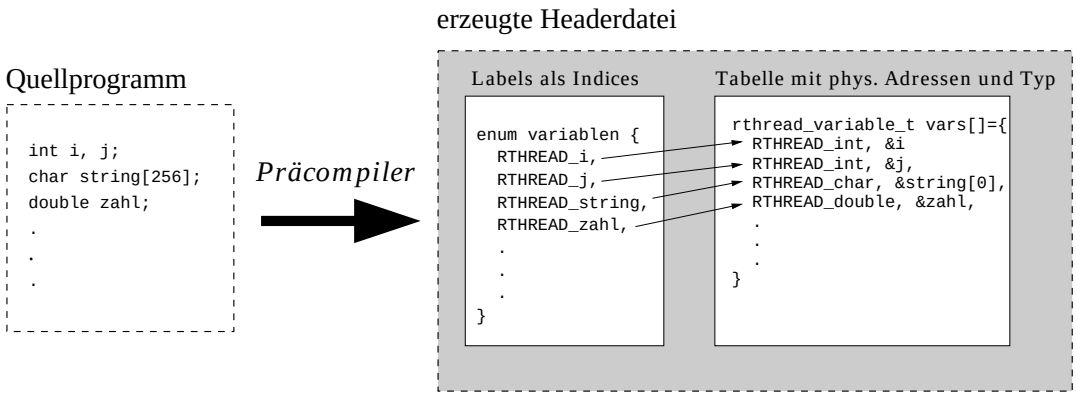

Abbildung 2: Aufgabe des Präcompilers

Tabelle sämtliche benötigten Informationen (im wesentlichen ist dies eben die Zuordnung zwischen globalem Identifikator und knotenlokalem Zeiger auf das Datum, sowie der Typ des Datums um im heterogenen System entsprechende Konvertierungen vorzunehmen) zu einem Datum ohne Suche erreichbar machen. Damit der Programmierer bei der Identifizierung nicht das Präfix `"RTHREAD_"` angeben muß, sorgt die Definition von entsprechenden Makros dafür, daß in den relevanten Funktionsaufrufen die Angabe des Variablennamens in den vollständigen Namen der Marke umgesetzt wird.

Die naive Anwendung des oben beschriebenen Vorgehens zur Vergabe von Identifikatoren auf Felder würde dazu führen, daß jedes Feldelement seinen eigenen Identifikator erhält. Ein erheblicher Speicher- und Verwaltungsaufwand wäre die Konsequenz. Um dies zu vermeiden wurde für Felder eine zweistufige Adressierung eingeführt. Der Präcompiler erzeugt für jedes Feld nur einen Identifikator. Die einzelnen Feldelemente werden durch diesen Identifikator sowie einen Offset zur Anfangsadresse des Feldes adressiert. Der Offset enthält dabei nicht den Abstand zum Anfang als physikalische Adresse, sondern den Abstand in Feldelementen. Damit ist die korrekte Funktionsweise in heterogenen Systemen, wo die gleichen Grunddatentypen auf verschiedenen Knoten verschiedene Größen haben können, wiederum gewährleistet. Da die Feldelemente logischerweise alle vom selben Typ sind, braucht auch dieser nur einmal gespeichert werden.

Mit dieser Adreßbildung können wir folgende angestrebten Ziele unterstützen: Das DSM-System bleibt durch die Verwendung eines (vom eigentlichen Compiler) unabhängigen, portablen Präcompilers selbst portabel. Heterogene Plattformen können durch die aus den Quelldateien gewonnenen Typinformationen und wegen der gewählten Granularität des DSM sehr gut und effizient unterstützt werden. Die Programmierung unseres Systems ist sehr nahe an der eines Shared-Memory-Systems angelehnt, da keine zusätzlichen Deklarationsfunktionen notwendig werden und globale Daten wie üblich über ihren Namen (*nicht* als Zeichenkette) identifiziert werden. Trotzdem ist der Zugriff auf die Informationen zu einem Datum durch die direkte Indizierung über die Identifikatoren schnell und nicht mit einer Suche in einer Tabelle verbunden.

2.4 Notwendige Einschränkungen der zugrundeliegende Programmiersprache

Spätestens bei der genaueren Betrachtung des Präcompilers muß die Sprache, mit der das DSM-System programmiert werden soll, festgelegt werden. Die Wahl der zugrundeliegenden Programmiersprache fiel auf *C*. Der Grund hierfür liegt in der großen Verbreitung von parallelen Anwendungen in dieser Sprache (um die Portierung bestehender Programme zu erleichtern) und die Unterstützung von Threads auf beinahe allen Systemen. Bezüglich der Verbreitung von parallelen Anwendungen kam außer *C* vor allem Fortran in die engere Auswahl. Allerdings gibt es derzeit weder für Fortran noch für C++ Threads, die von beinahe jedem System unterstützt werden, wie dies bei Pthreads und *C* der Fall ist. Bei Java wiederum sind zwar Threads bereits im Sprachumfang enthalten, aber es existieren kaum parallele Programme, die sich mit praktischen, rechenintensiven Algorithmen auseinandersetzen. Wegen der bislang deutlich schlechteren Laufzeiten von Java-Programmen wird dies kurz- und mittelfristig wohl auch so bleiben. Daher gaben wir der Sprache *C* den Vorzug.

Der Rthreads Präcompiler ist in der Lage sämtliche *C*-Datentypen, die sich innerhalb der *analysierten* Quelldateien zu einem Grunddatentyp auflösen lassen, zu erkennen (auch Felder). Der Präcompiler ist nicht in der Lage *C*-Präprozessoranweisungen auszuwerten, was z.B. im Fall von bedingten Übersetzungen zu Problemen führen kann. Diese Probleme lassen sich aber durch die Übergabe der zu den ursprünglichen Quelldateien gehörigen *C*-Präprozessorausgaben an den Präcompiler beseitigen.

Eine Einschränkung an die *C*-Quellprogramme ist der Ausschluß von Zeigern aus dem DSM.[5] Zeiger können in einem DSM-System mit gemeinsamen Datenobjekten durch die veränderte Adreßbildung kaum sinnvoll gehandhabt werden. Bei einem Schreib- oder Lesezugriff auf einen Zeiger müßten auch die von dem Zeiger referenzierten Daten aktualisiert werden. Da Zeiger in *C* aber sogar während der Ausführung eines Programms auf verschiedene Datentypen zeigen können, ist die korrekte Übertragung der Daten in einem heterogenen Netz sehr schwer sicherzustellen. Da es in diesem Zusammenhang noch viele weitere ungeklärte und unklärbare (einem Zeiger im globalen Datenraum wird die Adresse eines lokalen Datums auf einem Knoten zugewiesen, die auf den anderen Knoten natürlich ungültig ist) Fragen gibt, unterstützt das hier vorgestellte System im allgemeinen keine Zeiger im DSM. Der einzige Fall, in dem Zeiger im DSM aus unserer Sicht zwingend notwendig sind, ist die Bereitstellung von Speicher in einer zur Laufzeit bestimmten Größe im DSM. Daher wird diese Technik von unserem System explizit durch eigene Methoden unterstützt, die hier aber nicht weiter beschrieben werden sollen.

[5] Dies betrifft allerdings nur Zeiger, die im DSM-System global verwendet werden sollen. Es ist möglich, knotenlokal Zeiger auf Daten des DSM (bzw. auf die lokale Kopie eines Datums) zu verwenden, wobei der Abgleich der lokalen Kopie mit dem DSM wiederum über das Datum selbst und nicht über den lokalen Zeiger vorgenommen werden muß. Es ist jedoch nicht möglich, im DSM Zeiger auf andere Daten (sinnvollerweise ebenfalls innerhalb des DSM) zu halten.

3 Zugriff auf den virtuell gemeinsamen Speicher

3.1 Zugriffsfunktionen

Wie Abbildung 1 bereits veranschaulichte, arbeiten sämtliche Knoten zunächst auf ihren lokalen Kopien der Daten des DSM, d.h. auf ihren globalen Variablen. Dies erlaubt den natürlichen Umgang mit den Daten des DSM ähnlich zur Shared-Memory-Programmierung. Zu verschiedenen Zeitpunkten ist aber natürlich der Abgleich von lokalen Kopien mit dem DSM notwendig. Dieser Abgleich wird von den Knoten selbst *ausgelöst*, er resultiert jedoch nicht in einem direkten Datenaustausch des aufrufenden Programmes mit dem DSM — es wird lediglich der Abgleich der betreffenden Daten durch unser DSM-System veranlaßt. Im Falle einer Leseoperation braucht sich das Programm selbst also nicht um die Speicherung eines gelesenen Wertes an eine bestimmte Adresse kümmern. Vielmehr wird zum Lesen lediglich das betreffende Datum über seinen Identifikator spezifiziert, nach dem Ende der Leseoperation enthält die entsprechende globale Variable als Teil des DSM einen konsistenten Wert. Die Verwendung der Funktionen, die hierfür vom Präcompiler eingesetzt werden, verdeutlicht das folgende Codebeispiel. In einem kritischen Abschnitt werden zwei Variablen aktualisiert, eine davon wird danach verändert:

```
rthread_mutex_lock(&job_lock);
rthread_r(step),
rthread_rflush(next_job),
my_job = next_job;
next_job += step,
rthread_wflush(next_job);
rthread_mutex_unlock(&job_lock);
```

Sämtliche Vorarbeiten, die für die Verwendung der textuellen Namen der Variablen in den Schreib-/Leseaufrufen benötigt werden, werden vom Präcompiler automatisch erzeugt. Die *rthread_r** Funktionen dienen dabei zum Lesen des aktuellen Wertes des Datums im DSM in die lokale Kopie. Die anderen Funktionen aktualisieren das Datum im DSM mit dem Wert der lokalen Kopie. Dabei erlauben *rthread_r()* bzw. *rthread_w()* die Markierung zum Lesen bzw. Schreiben. Bei der Markierung wird das Datum lediglich für die entsprechende Operation vorgemerkt, eine Speicherung des gegenwärtigen Wertes einer lokalen Kopie zum späteren Schreiben in den DSM erfolgt nicht! Die beiden *flush*-Funktionen sorgen dann blockierend für den sofortigen Abgleich (über das Verbindungsnetzwerk) aller bis dahin zum Lesen bzw. Schreiben markierten (und bislang noch nicht abgeglichenen) Daten. Durch „sofortigen Abgleich“ garantiert das hier vorgestellte DSM-System dabei in der Terminologie aus [Gil93] folgendes:

- Nach der Beendigung eines Aufrufs von *rthread_wflush()* gelten sämtliche Schreibzugriffe auf zuvor markierte Daten (s.o.) als ausgeführt.

- Nach der Beendigung eines Aufrufs von *rthread_rflush()* gelten sämtliche Lesezugriffe auf zuvor markierte Daten (s.o.) als global ausgeführt.

Dabei gelten als Lese- oder Schreibzugriffe anderer Prozessoren nur deren Verwendung von *flush*-Operationen (und nicht *rthread_r()*- oder *rthread_w()*-Operationen, die nur der Markierung dienen). Die Garantie bezieht sich also lediglich auf die Abfolge der *flush* Operationen.

Der Zugriff auf den DSM erfolgt mit den obengenannten Funktionen völlig transparent, Typ- oder Lokationsinformationen zu einem Datum sind nicht notwendig, es genügt die Identifikation des Datums.

3.2 Effizienter Einsatz der Zugriffsfunktionen

Die oben angegebene Schnittstelle läßt jeglichen Raum für Optimierungen durch den Programmierer. Sie erlaubt das Gruppieren von DSM-Zugriffen durch aufeinanderfolgende Markierung mehrerer Daten zu einer Netzwerktransaktion, die durch den Aufruf einer der beiden *rthread_*flush()* Aufruf ausgelöst wird. Die Granularität der Daten im DSM bezüglich der Netzwerkzugriffe wird dadurch stufenlos an die jeweiligen Bedürfnisse der Applikation angepaßt. Damit kann eines der häufigsten Probleme von DSM-Systemen, nämlich das *häufige* Verschicken *kleiner* Nachrichten, das in nahezu allen Netzwerken wegen der Latenzzeiten und dem Protokoll-Overhead sehr viel langsamer als das einmalige Verschicken einer größeren Nachricht ist, gelindert werden. Zugleich kann der Programmierer selbst bestimmen, wann ein Abgleich von Daten wirklich notwendig ist. Ein DSM-System kann diese Aufgabe selbständig niemals so gut lösen, wie der Programmierer selbst, da es die Anforderungen des Algorithmus bezüglich der Datenkonsistenz nicht kennt und deswegen eine defensive Strategie einschlagen muß.

Es darf natürlich nicht verschwiegen werden, daß dadurch auch eine gewisse Last auf den Programmierer abgewälzt wird. Die in dieser Arbeit beschriebene Schnittstelle soll jedoch zunächst lediglich eine Basis darstellen. Darauf aufbauend können auch Automatismen implementiert werden, die dem Programmierer diese Last zum Datenabgleich, z.B. im Sinne der früher vorgestellten Konsistenzmodelle, abnehmen. Ein Leitfaden, wie mit unserem System Programme erstellt werden können, die sich einem bestimmten Konsistenzmodell entsprechend verhalten, findet sich in Abschnitt 4.2. Darüber hinaus muß es jedoch möglich bleiben, an einem Programm eigenhändige Optimierungen vorzunehmen, um letztendlich eine befriedigende Effizienz zu erreichen. Unsere Idealvorstellung ist es hierbei, daß durch die Einführung von obengenannten Automatismen die Erstellung eines lauffähigen Programms für unser System mit sehr wenig Aufwand verbunden sein sollte. Falls die dadurch erreichte Effizienz nicht ausreichend sein sollte, muß dem Programmierer aber die Möglichkeit gegeben werden, sein Wissen über den Algorithmus einzubringen und so die Netzwerkzugriffe zu minimieren. Daher muß die hier vorgestellte Schnittstelle den Raum für solche Optimierungen lassen. Darauf aufbauend werden dann Mechanismen implementiert, die sozusagen der „Bequemlichkeit“

des Programmierers und damit der schnelleren Programmentwicklung dienen. Mit dieser Strategie versuchen wir ein häufiges Manko anderer DSM-Systeme anzugehen, mit denen es zwar leicht ist, ein Programm zu erstellen, die jedoch keine Möglichkeit für spätere Optimierungen lassen. Gedanken zum Einbringen von Konsistenzmodellen und eine genauere Betrachtung, wie sich die eben vorgestellten Funktionen verwenden lassen, um bekannte Konsistenzmodelle nachzubilden, finden sich erst in Abschnitt 4.2, da dazu das Verständnis der Synchronisationsoperationen (siehe Abschnitt 4) des hier vorgestellten Systems vonnöten ist.

3.3 Implementation der Zugriffsfunktionen

Bei der Implementation der obengenannten Schreib-/Leseoperationen ist offensichtlich Netzwerkkommunikation mit anderen Knoten notwendig. Auch hier besteht wieder die Gefahr, Portabilität durch die spezielle Implementation auf bestimmten Netzwerkwelten (z.B. TCP/IP oder aber auch IPX) zu verlieren. Natürlich wäre es möglich, eine Implementierung für einige dieser Welten vorzunehmen. Dadurch würden aber auf jeden Fall Zielsysteme spezieller Architektur (und nicht nur solche), wie z.B. die CRAY T3E oder auch die IBM/SP-Serie, ausgeschlossen oder zumindest nur sehr ineffizient unterstützt. Allen heute in der Praxis verwendeten vernetzten Systeme ist jedoch gemeinsam, daß die bekannten Message-Passing-Systeme PVM oder MPI, oder auch das verteilte Betriebssystem DCE, auf deren spezielle Netzwerkarchitektur angepaßt wurden. Jeder Hersteller von Parallelrechnern ohne gemeinsamen Speicher liefert heute eine speziell für die Kommunikation in diesem Rechner optimierte PVM- und/oder MPI-Version aus. Daher haben wir uns entschlossen, die gesamte Netzwerkkommunikation über bestehende Systeme abzuwickeln. Dies dient in erster Linie der Portabilität. Gleichzeitig ist unser System dadurch aber auch in der Lage zum einen auf *jedem* System eine effiziente Implementierung zu benutzen, zum anderen von Weiterentwicklungen des zugrundeliegenden Systems zur Kommunikation zu profitieren. Ein weiterer Vorteil, der unseren Zielen sehr entgegenkommt, ist, daß diese Systeme heterogene Netze von sich aus unterstützen. Zusammen mit den vom Präcompiler gewonnenen Typinformationen arbeiten die oben genannten Schreib-/Leseoperationen damit problemlos in heterogenen Umgebungen. Implementationen unseres Systems liegen derzeit für PVM, DCE, MPI und demnächst für Active Messages vor — damit dürfte sich kaum ein System finden, das keines dieser Kommunikationspakete unterstützt.

4 Nebenläufigkeit und Synchronisation

4.1 Einführung einer Hierarchie aus Pthreads und Rthreads

Um wirklich mit einem DSM-System programmieren zu können, müssen Mechanismen zur Beschreibung von Nebenläufigkeit zur Verfügung gestellt werden. Aus heutiger Sicht ist es nicht üblich den Schutz von kritischen Bereichen ohne explizite Synchronisation

zu implementieren. Daher ist es auch sinnvoll, ein Konzept zur Synchronisation der parallelen Teile eines Programms durch das DSM-System zur Verfügung zu stellen.

Für das hier vorgestellte DSM-System war es eines der Ziele, Multithreading zuzulassen bzw. sogar explizit zu unterstützen. Bei der Wahl eines geeigneten Thread-Modells fiel die Entscheidung zugunsten der POSIX Threads. POSIX Threads bieten eine standardisierte Programmierschnittstelle für Threads, die sich derzeit auf allen Betriebssystemplattformen durchzusetzen scheint. Durch die Unterstützung dieses Standards können wir wieder die angestrebte Portabilität in zweierlei Sinn erzielen: Zum einen gibt es heute bereits viele bestehende POSIX-Thread-Programme, die damit leicht zum Ablauf in unser System modifiziert werden können. Außerdem ist durch die Verwendung dieses Standards auch gewährleistet, daß unser System selbst, das sich auf diese Threads[6] abstützt, auf möglichst viele Plattformen übertragen werden kann.

Durch die Verwendung von Pthreads ist es zunächst möglich, Parallelität im Sinne von Multithreading und Synchronisation innerhalb eines Prozesses einzuführen. Die Beschreibung von Nebenläufigkeit und Synchronisation wird für unser DSM-System auch auf einer höheren Ebene — zwischen unabhängigen, vernetzten Knoten eines verteilten Systems — benötigt. Es liegt nahe, dieselbe Programmierschnittstelle, wie wir sie von Pthreads kennengelernt haben, auch für diese Ebene zu verwenden. Auf diese Art und Weise kann der Programmierer Nebenläufigkeit und Synchronisation zwischen mehreren Knoten eines verteilten Systems nach demselben Paradigma beschreiben, wie er es von der Thread-Programmierung innerhalb eines Prozesses gewohnt ist.

Um die Programmierung unseres Systems in der „Sprache“ der Pthreads zu erlauben, werden die entsprechenden Funktionen der Pthreads übertragen. Zu jeder der relevanten *pthread_*()*-Funktionen wurden bedeutungsgleiche *rthread_*()*-Funktionen entwickelt. Damit erlauben wir die Beschreibung der Parallelität zwischen mehreren Knoten unseres Systems via *rthread_create()*. Der Aufruf dieser Funktion führt zum Start eines Prozesses oder Rthreads auf einem selbständigen Knoten, der mit der Ausführung des angegebenen Unterprogramms beginnt. Dieses Vorgehen gab unserem System seinen Namen: Rthreads. „R“ steht für „remote“ und „Threads“ bezieht sich auf die Ähnlichkeit zu Pthreads bezüglich der Programmierschnittstelle.

Durch diese Modellierung wird eine Hierarchie eingeführt: Jeder Knoten des DSM-Systems ist in der Lage einen neuen Prozeß (Rthread), der verteilt abläuft, auf einem anderen Knoten zu starten (*rthread_create()*). Innerhalb eines Knotens können außerdem mehrere Pthreads parallel ablaufen. Die Pthreads eines Knotens können auch auf die Variablen des DSM zugreifen und für den Abgleich von lokalen Kopien mit dem DSM sorgen. Für die Synchronisation von Pthreads eines Prozesses stehen nach wie

[6] Genauer gesagt werden die Threads der POSIX 1003.4a Draft 4 Programmierschnittstelle für leichtgewichtige Prozesse (Threads) verwendet. Dieser Draft wird oftmals auch mit DCE Threads bezeichnet, da eine Implementation dieses Drafts in DCE integriert ist. Die Implementation unseres Systems hält sich an diese Thread-Variante, da der endgültige Standard für Threads (POSIX 1003.1c-1995) leider bis heute noch nicht auf allen Betriebssystemen verfügbar ist. Eine Umstellung auf den POSIX Standard ist allerdings nicht allzu aufwendig, weshalb wir im folgenden trotzdem den Begriff POSIX Threads oder Pthreads verwenden.

vor die entsprechenden *pthread_*()* Funktionen zur Verfügung, die auf Variablen dieses Prozesses arbeiten. Sobald ein beliebiger Pthread eines Knoten jedoch *rthread_*()*-Funktionen verwendet, wird dadurch die Synchronisation auf der Ebene der Rthreads über Synchronisationsdaten des DSM veranlaßt. Die Pthreads innerhalb eines Rthreads sind gleichberechtigt, z.B. kann jeder von ihnen mittels *rthread_create()* einen neuen Rthread starten. Ebenso sind die Rthreads untereinander gleichberechtigt. Zur Synchronisation mit anderen Rthreads, müssen die Pthreads jedoch die *rthread_*()* Funktionen auf Synchronisationsdaten im DSM anwenden. Die neue Situation wird durch Abbildung 3 veranschaulicht.

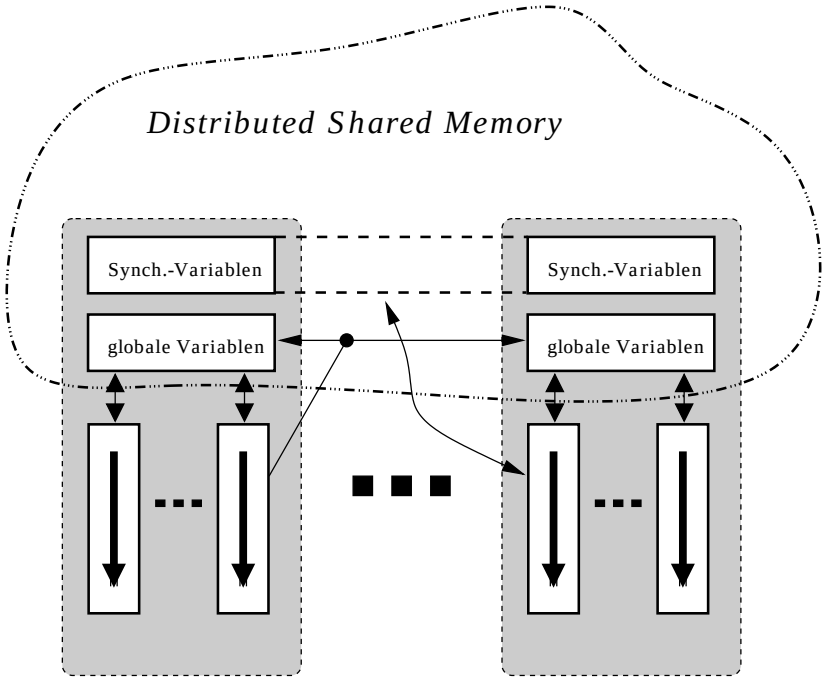

Abbildung 3: Aufbau von Rthreads

Im Vergleich zu Abbildung 1 sehen wir zunächst, daß jeder teilnehmende Knoten jetzt seinerseits wieder aus mehreren parallel ablaufenden Kontrollfäden (Pthreads) bestehen kann. Jeder dieser Pthreads kann wieder wie gewohnt auf die globalen Variablen zugreifen und auch den Abgleich mit dem DSM veranlassen.

Da durch den expliziten Aufruf von Synchronisationsfunktionen im Programmiermodell der Pthreads sowieso eine Sonderstellung der Synchronisationszugriffe eingeführt wird, behandeln wir Synchronisationsdaten anders als die restlichen Daten des DSM. Die Betrachtungen über Konsistenzmodelle haben gezeigt, daß alle schwächeren Konsistenzmodelle, die in modernen Systemen Anwendung finden, eine solche Unterscheidung vornehmen. Die Begründung hierfür ist, daß in heutigen parallelen Programmen in der Praxis ohnehin explizite Synchronisationsaufrufe angegeben werden. Dies hat sich auch für die von uns verwendeten Pthreads als gültig erwiesen. Die angesprochenen Konsistenzmodelle schlagen für Synchronisationszugriffe entweder sequentielle Konsistenz (im

Falle des schwachen Konsistenzmodells) oder Prozessorkonsistenz (im Falle der Release- oder Entry-Konsistenz) vor.

Offenbar macht es keinen Sinn für die Synchronisationszugriffe noch schwächere Modelle zu verwenden, da dann die Intention durch die Synchronisation die anderen Speicherzugriffe zu ordnen, verletzt würde. Wir haben uns aus folgenden Überlegungen heraus für die Anwendung der sequentiellen Konsistenz entschieden: Erheblicher Geschwindigkeitsgewinn wird durch die Prozessorkonsistenz nur erzielt, wenn man das Überholen von Lese- bzgl. Schreibzugriffen nach Gharachorloo [GL+90] zuläßt. Für Synchronisationszugriffe bedeutet dies im speziellen Fall der von uns verwendeten *lock()*- (Lese- und evtl. Schreibzugriff auf dasselbe Datum) und *unlock()*-Funktionen (Schreibzugriff) das Verschieben der *lock()*-Aufrufe nach vorne relativ zu den *unlock()*-Aufrufen. Dadurch werden die kritischen Bereiche vergrößert, was zur Einschränkung der Parallelität führt. Geschwindigkeitsgewinne durch die Verwendung der Prozessorkonsistenz für Synchronisationszugriffe wären daher in unserem Fall durchaus zweifelhaft. Da die sequentielle Konsistenz außerdem wesentlich leichter zu verstehen ist und auch dem entspricht, was der Programmierer vom Umgang mit Pthreads kennt, wenden wir für Synchronisationszugriffe diese sequentielle Konsistenz an. Dies erlaubt eine einfache Realisierung und Veranschaulichung der Synchronisationsoperationen, die ebenfalls in Abbildung 3 symbolisiert wird. Beim Zugriff auf Synchronisationsdaten des DSM arbeiten die Rthreads nicht auf lokalen Kopien, sondern führen ihren Zugriff sofort über das Netzwerk zum beherbergenden Eigentümer aus. Dort werden die Synchronisationsfunktionen auf die Pthread-Pendants zurückgespielt, die dann einfach auf den prozeßglobalen Synchronisationsvariablen des Eigentümers arbeiten. Durch deren Verwendung wird die Serialisierung der Anfragen auf dem Eigentümer erreicht. Da es für jedes Synchronisationsdatum natürlich genau einen Eigentümer gibt (und Kopien davon nicht verwendet werden), ist dadurch auch die globale Sequenz der Anfragen an Synchronisationsdaten gewährleistet, die die sequentielle Konsistenz fordert. Natürlich sind dann auch keine weiteren Funktionen zum Abgleich der Synchronisationsdaten, wie im Fall der anderen Daten des DSM, mehr nötig.

4.2 Bemerkungen zur Konsistenz

Die Schreib-/Leseoperationen in der bisher dargestellten Form arbeiten nicht nach einem vordefinierten Konsistenzmodell. Die Konsistenz der Daten muß durch das Programm selbst an den notwendigen Stellen durch den Aufruf der *rthread_*flush()*-Funktionen gesichert werden. Erst nach der Ausführung der jeweiligen Funktion garantiert das Rthreads-System, daß die Schreib- bzw. Leseoperation global ausgeführt wurde. Obwohl kein spezielles Konsistenzmodell explizit unterstützt wird, ist es für den Programmierer recht einfach Programme zu entwickeln, die nach einem bestimmten solchen Modell arbeiten. Werden z.B. vor jeder Verwendung einer globalen Variablen im Programm als R-Value ein *rthread_rflush()* und nach jeder Verwendung als L-Value ein *rthread_wflush()* bezüglich der verwendeten Variablen eingefügt, so erhält man ein Programm, das sich wie in einem sequentiell konsistenten System verhält: Jeder Datenzugriff wird sofort

über das Netzwerk beim Eigentümer des Datums ausgeführt. Dort wird die Serialisierung durch die nacheinander eingehenden Netzwerkaufrufe erreicht und daraus entsteht die einheitliche Sicht der Datenzugriffe für alle teilnehmenden Prozessoren.

Da das von uns zugrundegelegte Programmiermodell explizite Synchronisationsaufrufe verlangt, wird durch die Verwendung von starken Konsistenzmodellen (also solchen, die Synchronisationszugriffe nicht unterscheiden) unnötig Effizienz verschenkt. Ausgehend von dem Programm, das durch Einfügen der expliziten Schreib-/Leseoperationen entstanden ist, lassen sich Programme, die nach schwächeren Konsistenzmodellen arbeiten einfach durch Verschieben (und Zusammenfassen) dieser Operationen erreichen. Unser Programmiermodell nimmt bereits eine Unterscheidung der Synchronisationsoperationen nach *acquire* und *release* vor (dabei interpretiert man das Erreichen eines *rthread_cond_wait()* als *release* und das Verlassen desselben als *acquire*). Alle neueren Untersuchungen zeigen, daß die *lazy-release*-Konsistenz die besten Laufzeiten erzielt. Da sich dieses Modell außerdem bezüglich der Synchronisationsoperationen harmonisch in unser System einfügt und mit Rthreads auch leicht zu realisieren ist, beschreibt der folgende Abschnitt die Entwicklung eines *lazy-release*-konsistenten Programms mit Rthreads. Die Realisierung anderer Modelle sollte aus dieser Beschreibung leicht nachzuvollziehen sein, weswegen wir darauf nicht mehr weiter eingehen.

Die folgende Beschreibung soll auch zeigen, daß die Realisierung eines *festen* Konsistenzmodells mit Rthreads ebenfalls leicht machbar ist. Als großen Vorteil sehen wir es jedoch, daß der Programmierer nicht gezwungen ist, sich fest an ein solches Schema zu halten. In der Praxis könnte das im folgenden erläuterte Vorgehen also ein erster Schritt zur Erzeugung eines effizienten und korrekten (im Sinne der sequentiellen Konsistenz) Programmes sein, ohne die Details des Algorithmus zu kennen. Eine weitere Optimierung durch Kenntnis des Algorithmus ist durchaus möglich und auch erwünscht.

4.3 Entwicklung eines lazy release konsistenten Programms

Zunächst muß durch folgende Kategorisierung der Zugriffe die Grundlage dafür gelegt werden, daß sich ein *lazy-release*-konistentes Programm gleich verhält wie ein sequentiell konsistentes: Wie schon früher erwähnt, hat Gharachorloo in [GL+90] formal bewiesen, daß sich ein sogenanntes *properly-labeled*-Programm in einem *release*-konsistenten System genauso verhält, wie in einem mit sequentieller Konsistenz. *Properly labeled* bedeutet in diesem Zusammenhang, daß die Zugriffe auf die Daten des DSM richtig charakterisiert und dementsprechend synchronisiert werden (eine formale Definition findet sich in [GL+90]). Das Ergebnis entspricht in etwa einem parallelen Programm, indem es durch ausreichende Synchronisation nicht zu Wettkampfbedingungen zwischen mehreren Prozessoren kommt. Eine korrekte Synchronisierung vorausgesetzt, bedeutet dies angewandt auf die *release*-Konsistenz bezüglich der Charakterisierung laut Gharachorloo, daß synchronisierende *acquire*- und *release*-Zugriffe, sowie konkurrierende und nichtkonkurrierende Datenzugriffe unterschieden werden müssen, um ein *properly-labeled*-Programm zu erhalten. Sämtliche Synchronisationszugriffe sind in unserem Programmiermodell bereits durch den expliziten Funktionsaufruf erkennbar und werden vom

Rthreads-System sequentiell konsistent behandelt. Also muß der Programmierer lediglich für die der *release*-Konsistenz entsprechende Behandlung der nicht-synchronisierenden Datenzugriffe sorgen, um ein Programm zu erhalten, das sich wie ein sequentiell konsistentes verhält. Dabei sind nach der Definition von *release*-Konsistenz konkurrierende und nicht-konkurrierende Zugriffe zu unterscheiden. Konkurrierende Zugriffe sind hierbei potentiell gleichzeitige Zugriffe verschiedener Prozessoren auf dasselbe Datum, wobei mindestens ein Zugriff schreibend ist. Nicht-konkurrierende Zugriffe sind in der Konsequenz nur lesende Zugriffe, Zugriffe auf unterschiedliche Daten oder Zugriffe, die durch Synchronisation vor gleichzeitigem Zugriff geschützt sind. Sind konkurrierende und nicht-konkurrierende Zugriffe erkannt, so können sie entsprechend der *release*-Konsistenz folgendermaßen behandelt werden:

Nicht-konkurrierende Zugriffe: Die Definition von der *release*-Konsistenz bedeutet für nicht-konkurrierende Lese- und Schreibzugriffe, daß diese nach der Programmordnung nicht *vor* ein *acquire* und nicht hinter ein *release* verschoben werden dürfen. In der Konsequenz ist es erlaubt, sämtliche Leseoperationen an den Beginn eines kritischen Bereichs und sämtliche Schreiboperationen ans Ende eines kritischen Bereichs zu verschieben, ohne die *release*-Konsistenz zu verletzen. Damit können alle Lese- bzw. Schreiboperationen jeweils gebündelt am Anfang bzw. am Ende eines kritischen Bereichs in *einem* Netzwerkaufruf vorgenommen werden.[7] Man markiert also direkt nach dem *acquire* alle im kritischen Bereich als R-Value verwendeten Variablen zum Lesen und führt die Leseoperation danach mit einem *rthread_rflush()* aus. Analog verfährt man mit den als L-Value verwendeten Variablen am Ende des kritischen Bereichs mit den Schreiboperationen.

Konkurrierende Zugriffe: Nicht-synchronisierende (synchronisierende Zugriffe brauchen hier nicht mehr berücksichtigt zu werden), konkurrierende Datenzugriffe müssen nach der Definition von *release* Konsistenz untereinander prozessorkonsistent nach Gharacharloo sein. Dies bedeutet, daß die Lesezugriffe bzgl. der Programmordnung nach oben verschoben werden dürfen und lediglich andere (konkurrierende) Lesezugriffe nicht überholen dürfen. Die Schreibzugriffe können nach unten verschoben werden ohne jedoch andere Schreibzugriffe zu überholen. Bei dem Verschieben der Lese- und Schreibzugriffe ist zu beachten, daß ein *acquire* als Lese- und Schreibzugriff gilt, und ein *release* als Schreibzugriff. Da die Synchronisationsoperationen ebenfalls konkurrierende Zugriffe sind, müssen sie hierbei berücksichtigt werden. Lesezugriffe dürfen also ein *release* nach oben überholen, Schreibzugriffe aber weder ein *acquire*, noch ein *release* nach unten.

Damit hat der Programmierer eine einfache Möglichkeit aus einem korrekt synchronisierten Pthread-Programm ein *release*-konsistentes Distributed-Shared-Memory-Programm

[7] Falls die zuvor zum Lesen markierten Daten von verschiedenen Eigentümern beherbergt werden, können genaugenommen mehrere Netzwerkaufrufe entstehen. Der schlimmste Fall, nämlich daß jedes markierte Datum von einem anderen Eigentümer beherbergt wird, ist jedoch sehr unwahrscheinlich. Zudem tritt selbst im schlimmsten Fall natürlich keine Verschlechterung im Vergleich zu einer naiven Implementation ein.

zu erzeugen, das sich ebenso verhält, wie das Pthread-Programm. Zunächst werden hierzu im Programmtext die *pthread_*()*-Synchronisationsfunktionen durch ihre entsprechenden Rthread-Pendants ersetzt. Dann müssen an den entsprechenden Stellen die expliziten Schreib-/Leseoperationen eingesetzt werden. Diese können anschließend nach dem oben vorgebenen Muster verschoben und gebündelt werden und werden mit einer *flush*-Operation abgeschlossen. Weitere Optimierungen durch den Programmierer, wie z.B. die Verwendung der speziellen Funktionen für Felder oder die Verwendung noch schwächerer Konsistenz, falls der Algorithmus dies für einige Daten zuläßt, können jederzeit vorgenommen werden.

Genau betrachtet entspricht die Konsistenz, welche durch ein nach oben angegebenem Muster entwickeltes Programm erreicht wird, genau der *lazy-release*-Konsistenz. Denn zum einen werden nur für die tatsächlich benötigten Variablen Leseanfragen eingefügt — die nicht benötigten Variablen in einem kritischen Bereich verbleiben also in einem inkonsistenten Zustand. Zum anderen werden die Variablen erst beim Betreten des kritischen Bereichs in einen konsistenten Zustand gebracht und nicht bereits wenn die Inkonsistenz durch einen Schreibaufruf eines anderen Prozessors erzeugt wird.

An dieser Stelle zeigt sich ein großer Vorteil der expliziten Funktionsaufrufe für Datenzugriffe im Quelltext. Die lauffähigen Programme können bezüglich des Netzwerktransfers im Vergleich zu Systemen, die die Zugriffe durch Compileränderungen einfügen und/oder ein festes Konsistenzmodell implementieren, wesentlich stärker optimiert werden — die Datenzugriffe können flexibel an den Algorithmus angepaßt werden.

4.4 Weitergehende Unterstützungen

Obwohl das Einfügen und Verschieben der Schreib-/Leseoperationen nicht allzu aufwendig ist, wird dem Programmierer hier in Zukunft noch mehr Arbeit abgenommen werden. Zur Zeit wird die bisher nur zur Extraktion der globalen Variablen verwendete Quellcodeanalyse durch den Präcompiler um das automatische Einfügen der Schreib-/Leseoperationen erweitert. Damit bräuchte der Programmierer lediglich noch das Verschieben nach dem oben angegebenen Muster vorzunehmen. Auch das Verschieben kann künftig entfallen, da sich auch eine direkte Unterstützung der *lazy-release*-Konsistenz[8] in Vorbereitung befindet. Durch deren Implementation wäre der Programmierer dann in der Lage, sich aus einem Pthread-Programm ein verteiltes DSM-Programm im wesentlichen automatisch erzeugen zu lassen. Trotzdem verbliebe ihm, im Gegensatz zu anderen Ansätzen, bei Rthreads immer noch die Möglichkeit Performancenachteile durch Optimierungen an den Schreib-/Leseoperationen im Quelltext wettzumachen.

[8] Wir haben uns im wesentlichen aus zwei Gründen für dieses Modell entschieden. Zum einen macht es wegen der expliziten Synchronisation in dem von uns gewählten Pthread-Programmiermodell keinen Sinn starke Konsistenzmodelle zu unterstützen. Von den schwächeren Konsistenzmodellen ist die LRK keinem der anderen Modelle unterlegen. In den meisten Fällen ist die *lazy-release*-Konsistenz den anderen sogar bezüglich Effizienz *und* Programmierbarkeit überlegen (siehe [KCDZ95] und [ACD+96]).

5 Programmentwicklung

Die vorangegangenen Abschnitte befaßten sich mit der Modellierung des Rthreads-Systems. Zum Schluß dieses Kapitel soll die Entwicklung eines Programmes für das Rthreads-System aus praktischer Sicht geschildert werden.

Die Programmierung selbst erfolgt in *C*.[9] Der einfachste und von uns vorgesehene Weg, um ein lauffähiges Rthread-Programm zu erstellen, ist es zunächst ein Pthread-Programm zu entwickeln. Dies bietet den großen Vorteil, daß das gesamte Debugging des Algorithmus' selbst sowie der notwendigen Synchronisationen lokal mit den proprietären Unterstützungen stattfinden kann. Zudem ist für den Progammierer, der die Thread-Programmierung bereits beherrscht, bis dahin kein zusätzlicher Lernaufwand notwendig. Bei der Entwicklung des Pthread-Programms sollte man allerdings bereits die Einschränkungen, die für ein Rthread-Programm einzuhalten sind, (im wesentlichen das Fehler von Zeigern innerhalb des DSM), berücksichtigen.

Um anschließend vom lokalen, vielfädigen Programm zum verteilten DSM Programm zu kommen, müssen zunächst die Datentypen und Synchronisationsaufrufe der Pthread-Bibliothek durch ihre Rthread-Pendants ersetzt werden. Da Rthreads mit POSIX Threads verträglich sind, sollte sich der Programmierer dafür zunächst Gedanken machen, welche Teilaufgaben aus dem parallelen Pthread-Programm sich für die verteilte Ausführung eignen. Für grobgranulare Abläufe werden die Pthread-Aufrufe in Rthread-Aufrufe umgewandelt, feingranulare Berechnungen können in der Anwendung unverändert mittels Pthreads gelöst werden.

Weiterhin müssen nach dem bereits angegebenen Schema Schreib-/Leseoperationen eingesetzt werden. Der Ablauf von Rthread-Programmen erfolgt nach dem Master/Slave-Paradigma, daher müssen zwei getrennte Programme erzeugt werden. Dabei wird das Slave-Programm aus dem bisherigen (Master-)Programm gewonnen, indem man aus sämtlichen globalen Variablen, den an ein *rthread_create()* übergebenen Funktionen oder solchen, die wiederum von diesen aufgerufen werden, und zuletzt einer vorgegebenen *main()*-Routine ein eigenständiges Programm erzeugt.[10]

Zum Ablauf des Rthread-Programms müssen die so erzeugten Programmdateien mit der Rthread-Bibliothek des zu verwendenden zugrundeliegenden Kommunikationssystems (derzeit MPI, PVM, DCE oder Active Messages) gelinkt werden. Das Starten von neuen Rthread-Prozessen ist nicht Bestandteil der Rthread-Implementierung selbst, sondern wird ebenfalls von dem gewählten Kommunikationssystem übernommen. Deshalb müssen Vorbereitungen, die das zugrundeliegende System verlangt (wie z. B. der Start der PVM-Konsole), vor dem Programmstart getroffen werden.

---

[9] Im Prinzip läßt sich das Konzept auch auf andere Sprachen, wie z.B. Fortran übertragen. Durch das Fehlen von Zeigern in Fortran ergäben sich sogar Vorteile bezüglich der Einschränkungen. Allerdings müßte dann zumindest der Präcompiler neu entwickelt werden. Als Grundvoraussetzung müssen für jede in Frage kommende Programmiersprache natürlich zunächst Pthreads vorhanden sein.

[10] Der gesamte in diesem Absatz beschriebene Vorgang kann künftig automatisch durch den erweiterten Präcompiler übernommen werden.

Wir schätzen es als großen Vorteil ein, daß beinahe die gesamte Programmentwicklung inklusive Debugging lokal mit den gewohnten Werkzeugen vorgenommen werden kann. Der Weg von einem so erstellten Pthread-Programm, das die Einschränkungen für Rthreads bereits erfüllt, ist kurz, und kann sogar automatisiert werden. Als weiteren markanten Vorteil sehen wir, daß eine spätere Optimierung eines lauffähigen Programms leicht im Quellcode vorgenommen werden kann.

6 Laufzeitergebnisse

Derzeit werden die Leistungsmerkmale der Rthread-Programmierschnittstelle untersucht. Im Vordergrund steht einerseits der Vergleich mit ähnlichen DSM-Systemen, andererseits wollen wir auch den Overhead, der durch das speichergekoppelte Programmierparadigma gegenüber den nachrichtenbasierten Grundsystemen eingebracht wird, ermitteln. Ein Fallbeispiel, das wir hier vorstellen wollen, ist die verteilte Berechnung einer Mandelbrotmenge. Zwar beschäftigen sich die aktuellen Auswertungen mit Standard-Benchmarks für Shared-Memory-Systeme (SPLASH [SWG91] und NAS [BBLS91] Benchmarks), doch die Untersuchungen sind leider noch nicht abgeschlossen.

Die Mandelbrotmenge wird von der Beispielapplikation wie folgt berechnet: Zunächst wird eine vorgegebene Anzahl von Rthreads auf den zur Verfügung stehenden Knoten gestartet. Die von den einzelnen Rthreads zu bearbeitende Aufgabe wird dabei nicht von vornherein zu gleich großen Teilen festgelegt, sondern die Arbeit wird nach dem Pool-Of-Tasks Konzept vergeben. Die Berechnung der 800×800 Pixel großen Farbmatrix wird in 50×50 große Teilstücke zerlegt. Die einzelnen Rthreads berechnen solange in einer Schleife jeweils das nächste Teilstück, bis die ganze Aufgabe bewältigt ist. Über die entfernte Speicherzugriffe (in Verbindung mit den notwendigen Synchronisationsoperationen) wird die Farbmatrix zwischen den beteiligten Knoten konsistent gehalten.

Als vergleichbares Distributed Shared Memory System haben wir für die folgende Meßreihe (vergleiche Abbildung 4) das DSM-System Adsmith [LKL96] verwendet. Desweiteren wurde eine speziell mit PVM entwickelte Variante des Algorithmus' entwickelt und in die Messung miteinbezogen. Während der Messreihe wurde die Anzahl der beteiligten Workstations bei gleichbleibender Problemgröße variiert. Bei beiden DSM-Systemen werden die Speicherzugriffe auf die Farbmatrix durch spezielle Zugriffsfunktionen realisiert, die die blockweise Übertragung eines Teilfeldes mit geringem Verwaltungsaufwand erlauben.

Das Laufzeitverhalten der PVM-basierten Mandelbrotvariante zeigt, daß das Problem in dem hier ausgewerteten Bereich sehr gut skaliert. In dieser nachrichtengekoppelten Implementierung ist die Kommunikation speziell auf die Bedürfnisse der Anwendung abgestimmt, eine Verdopplung der Rechenkapazität bewirkt jeweils in etwa eine Halbierung der Laufzeit.

Im Gegensatz dazu kann bei der auf Adsmith basierenden Variante nur bis zu einer Knotenzahl von acht Knoten eine Laufzeitverbesserung erzielt werden. Danach hält sich

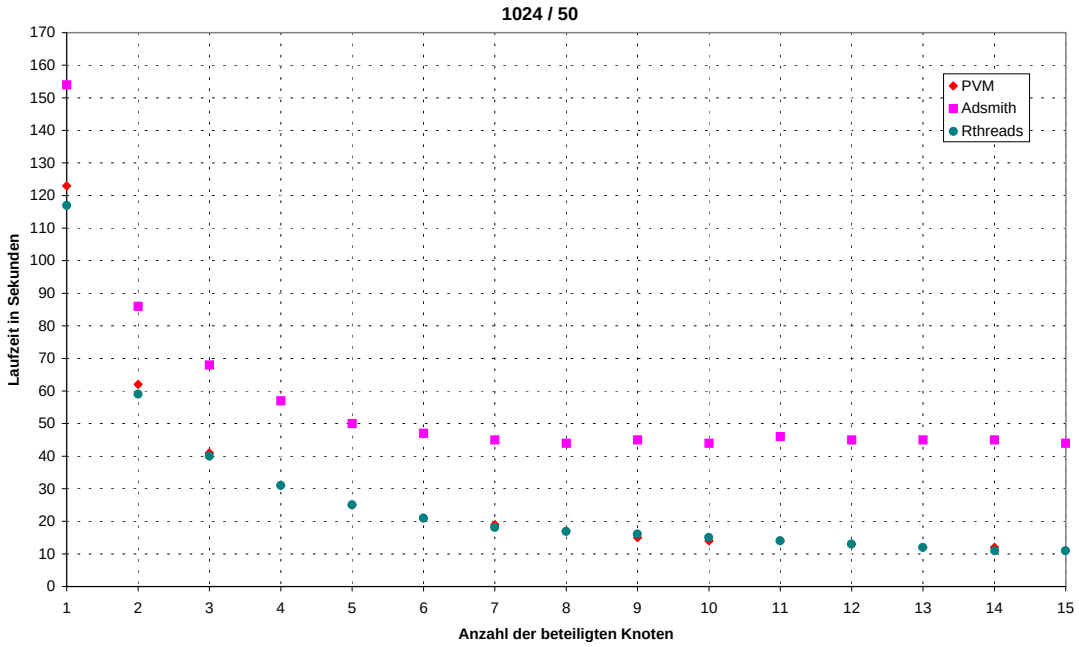

Abbildung 4: Laufzeitmessung mit Tiefe 1024 und Blockgröße 50×50

der Vorteil einer erhöhten Rechenkapazität mit dem gleichzeitig erhöhten Verwaltungs- und Kommunikationsbedarf des DSM-Systems ungefähr die Waage. Der einzige für uns ersichtliche Grund ist die etwas komplizierte Architektur von Adsmith: Adsmith wickelt die gesamte Verwaltung des DSM sowie die Zugriffe darauf über eigenständige DSM-Dæmons auf jedem Knoten ab, der notwendige Austausch der gemeinsam genutzten Speicherbereiche zwischen den verschiedenen Knoten erfolgt also immer über diese Zwischenstationen. Die lokale Kommunikation zwischen Dæmon und Programm erfolgt ebenso wie die Netzwerkkommunikation über PVM. Für jeden Schreibaufruf eines berechneten Teilstücks ist dadurch zunächst eine lokale Kommunikation mit dem Adsmith-Dæmon und eine weitere mit dem Empfängerknoten notwendig.

Die Vermutung, daß DSM-Systeme bei diesem Beispiel durch die zusätzlichen Synchronisationen im allgemeinen schlechtere Laufzeiten erzielen widerlegen die Messungen der Rthread-basierten Implementation: Sie erreichen innerhalb der gemessenen Knotenzahl dieselbe Performance wie die PVM-Version. Der durch die (im Vergleich zum reinen Message Passing) komplizierteren Schreib-/Leseoperationen und die zusätzlichen Synchronisationsoperationen zweifellos verursachte Overhead ist bei diesem einfachen Beispiel nicht meßbar. Die Architektur von Rthreads ermöglicht die Kommunikation zwischen den verschiedenen Rthreads einer verteilten Anwendung auf direktem Weg (mittels PVM). Eine eigene Programminstanz zur Verwaltung des DSM wird bei Rthreads nicht benötigt.

7 Zusammenfassung

Wir haben mit Rthreads ein DSM System entwickelt, das den virtuell gemeinsamen Speicher durch eine — bisher in diesem Zusammenhang nicht eingesetzte — automatische Analyse und Veränderung des Quellcodes durch einen eigenen Präcompiler realisiert. Da die Programmiermethodik der Pthreads übertragen wurde und die expliziten Schreib-/Leseoperationen durch den Präcompiler automatisch eingesetzt werden können, sind wir in der Lage POSIX 1003.4a-konforme vielfädige Programme (Pthread-Programme), wie sie für Multiprozessor-Workstations vielfach vorliegen, ohne Neuprogrammierung auf ein Rechnernetz zu transferieren. Die Pthread-Programme werden automatisch in Rthread-Programme transformiert. Die bisherigen Erfahrungen aus der Praxis zeigen, daß der Einstieg in die Programmierung mit Rthreads durch den hohen Bekanntheitsgrad von Pthreads sowie durch die Möglichkeit des lokalen Debuggings des (normalerweise zunächst entwickelten) Pthread-Programms leichter fällt als mit vergleichbaren Systemen.

Das Rthreads-System selbst besteht aus einem Präcompiler und Funktionsbibliotheken, wobei die Funktionsbibliotheken jeweils auf einem der weitverbreiteten Kommunikationssysteme PVM, MPI, DCE oder Active Messages aufsetzen. Damit ist das Rthreads-System auf allen Parallelrechnern und sogar auf heterogenen Workstation-Clustern lauffähig, sofern eines der oben genannten Pakete sowie POSIX Threads unterstützt werden.

Die bisherigen Laufzeitmessungen zeigten, daß durch das Aufsetzen auf PVM, MPI und DCE bei der reinen Datenübertragung nur geringer Overhead erzeugt wird. Darüberhinaus zeigt die Vergleichsmessung des vorherigen Abschnitts, daß sich der eingebrachte Overhead gegenüber einem (neu programmierten) Programm auf dem Grundsystem bei dem einfachen Beispiel der Berechnung von Mandelbrot-Mengen praktisch nicht messen läßt. Daß diese Erkenntnis für vergleichbare DSM Systeme durchaus nicht selbstverständlich ist, zeigen die von Adsmith erreichten Laufzeiten. Adsmith benutzt dasselbe zugrundeliegende Kommunikationssystem (PVM) und realisiert den gemeinsamen Speicher ebenfalls durch explizite Schreib-/Leseoperationen.

Derzeit steht die Implementierung eines erweiterten Rthread-Pakets kurz vor dem Abschluß. Es enthält insbesondere spezielle Unterstützung für kompliziertere, benutzerdefinierte Datentypen sowie für dynamisch allozierten Speicher innerhalb des DSM. Diese Mechanismen werden die bereits begonnene Portierung der SPLASH und NAS Benchmarks weiter erleichtern. Die Auswertung dieser Benchmarks wird uns weitere Ergebnisse über die Laufzeiten des vom Präcompiler automatisch erzeugten Code sowie den Aufwand zur Optimierung dieses Codes durch den Programmierer liefern. Beide Codes werden wir mit den Ergebnissen bestehender DSM-Systeme vergleichen, wobei die sehr guten Ergebnisse der bisher ausgewerteten einfachen Beispiele optimistisch stimmen. Von der gerade begonnenen Implementierung eines *lazy release* konsistenten Protokolls für Rthreads erhoffen wir uns, daß die Notwendigkeit der nachträglichen Optimierung durch den Programmierer weiter verringert wird.

Literatur

[ACD+96] Sarita Adve, Alan Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Second High Performance Computer Architecture Conference*, pages 26–37, February 1996.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[DCM95] I. Demeure, R. Cabrera-Dantart, and P. Meunier. Phosphorus: A Distributed Shared Memory System on Top of PVM. In *Proceedings of EUROMICRO'95*, pages 269–273, September 1995.

[Gil93] W. K. Giloi. *Rechnerarchitektur*. Springer-Verlag, Berlin, 1993.

[GL+90] Kourosh Gharachorloo, Daniel Lenoski, et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer architecture news*, 18(2):15–26, June 1990.

[KCDZ95] Pete Keleher, Alan Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29, September 1995.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Computer Architecture News*, 20(2), May 1992.

[LKL96] Wen-Yew Liang, Chun-Ta King, and Feipei Lai. Adsmith: An efficient object-based distributed shared memory system on PVM. *Proceedings of the 1996 International Symposium on Parallel Architecture (ISPAN 96)*, pages 173–179, June 1996.

[SWG91] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[ZSB94] Matthew Zekauskas, Wayne Sawdon, and Brian Bershad. Software write detection for a distributed shared memory. In *First Symposium on Operating Systems Design and Implementations*, 1994.