This paper appeared in: Informatik – Forschung und Entwicklung, Volume 7, Number 1, pages 1–13, January 1992.

Projekt Triton: Beiträge zur Verbesserung der Programmierbarkeit hochparalleler Rechensysteme

Michael Philippsen, Thomas Warschko, Walter F. Tichy und Christian Herter

Universität Karlsruhe
Institut für Programmstrukturen
und Datenorganisation
Postfach 6980

76128 Karlsruhe

email: philippsen@ira.uka.de

Zusammenfassung

Im Projekt Triton steht das Ziel der adäquaten Programmierbarkeit von Hochparallelrechnern mit mehreren tausend Prozessoren im Mittelpunkt. Dieses Ziel kann erreicht werden, wenn Programmiersprache und Hardware-Architektur gemeinsam entworfen und so aufeinander abgestimmt werden, daß eine Übersetzung der Sprache in effizienten Code möglich wird. Diesen Ansatz verfolgen wir im Projekt Triton.

1. Parallele Programmiersprache. Mit Modula-2*, einer Erweiterung von Modula-2, stellen wir Sprachmittel vor, die erforderlich sind, um parallele Algorithmen hochsprachlich, lesbar und portabel zu formulieren. Wir beschreiben deren effiziente Übersetzbarkeit für unterschiedliche moderne Parallelrechnerarchitekturen.

2. Parallele Hardware. Mit Triton/1 entwickeln wir neue Architekturansätze paralleler Hardware und realisieren diese exemplarisch. Durch die Vereinigung von SIMD- und MIMD-Prinzipien in einer SAMD („synchronous or asynchronous multiple data“) Architektur und den Einsatz eines hocheffizienten Kommunikationsnetzwerkes gelingt es, viele Schwächen heutiger Parallelrechner zu vermeiden und die Übersetzung maschinenunabhängiger paralleler Sprachen wie Modula-2* zu unterstützen.

Schlüsselwörter

Hochparallele Systeme, parallele Hochsprachen, Portabilität, optimierende Übersetzer, Perfect-Shuffle-Netzwerke, Skalierbarkeit, parallele Ein-/Ausgabe

Summary

The main objective of the Triton project is adequate programmability of massively parallel computer systems. This goal can be achieved by tightly coupling the design of programming language and parallel hardware. By combining these efforts, translation into efficient code becomes possible. This approach we took in the Triton project.

1. Parallel programming language. Modula-2*, an extension of Modula-2, is a programming language with elements for expressing arbitrary parallel algorithms in a high-level, portable, and readable way. We

present techniques for efficiently translating those elements for several modern parallel architectures.

2. Parallel hardware. With Triton/1 we develop novel architectural paradigms for parallel computers. By combining principles of SIMD and MIMD in a SAMD, i.e., synchronous or asynchronous multiple data architecture, and by designing a highly efficient communications network, Triton/1 overcomes various deficiencies of current parallel hardware and supports parallel languages such as Modula-2*.

Key words

massively parallel systems, parallel high-level languages, portability, optimizing compilers, perfect shuffle networks, scalability, parallel I/O

Computing Reviews Classification

D.3.3, D.3.4, C.1.2

1 Einleitung

Im Projekt Triton steht das Ziel der adäquaten Programmierbarkeit von Hochparallelrechnern mit mehreren tausend Prozessoren im Mittelpunkt. Gute Programmierbarkeit bedeutet dabei,

- daß Programme ausgehend von einem Lösungsalgorithmus leicht und schnell zu formulieren sind,
- daß die erstellte Software portabel, lesbar und wartbar ist und schließlich
- daß eine ausreichend effiziente Ausführung von Algorithmen möglich ist.

Die bereits kommerziell verfügbaren Maschinen wurden fast ausschließlich basierend auf technischen Machbarkeitsüberlegungen entwickelt. Erst nachträglich wurden Programmiermodelle ergänzt, deren wesentliches Ziel es war, die theoretische Rechen- und Kommunikationsleistung der Maschine möglichst weitgehend praktisch erreichbar zu machen.

Während die ersten Nutzer von Parallelrechnern bereit waren, sehr hohen Programmieraufwand hinzunehmen, um Höchstleistungen bei der Lösung spezifischer Aufgaben zu erreichen, gewinnt mit zunehmender Verfügbarkeit dieser Anlagen die Programmierbarkeit an Bedeutung. Gefordert wird, daß Software wegen der hohen Entwicklungskosten zwischen unterschiedlichen Maschinenarchitekturen portabel und wartbar ist. Ferner werden von Programmiersprachen Sprachmittel verlangt, die genügend ausdrucksstark sind, um beliebige parallele Algorithmen gut lesbar formulieren zu können, und sich nicht auf bestimmte Problemklassen spezialisieren.

Die parallele Informatik wiederholt zur Zeit viele Entwicklungsstufen, die auch die sequentielle Informatik in den letzten vier Dekaden durchlief. Die Bereitschaft wächst, für Komfort bei der Programmierung Leistungseinbußen in der Ausführungszeit zu tolerieren. Durch verbesserte Hardware-Architekturen, Betriebssysteme, Programmiersprachen und Übersetzertechnologien wird deshalb maschinenabhängiges Programmieren von Parallelrechnern bald die gleiche Bewertung erfahren, die Assemblerprogrammierung für sequentielle Rechner bereits heute hat.

Ähnlich wie in der sequentiellen Informatik kann das Ziel der guten Programmierbarkeit paralleler Rechnersysteme nur dann erreicht werden, wenn Programmiersprache und Hardware-Architektur gemeinsam entworfen und so aufeinander abgestimmt werden, daß eine gute Übersetzung der Sprache in effizienten Code möglich wird. Diesen Ansatz verfolgen wir im Projekt Triton.

1. Parallele Programmiersprache. Wir beschreiben die Schwachpunkte der in der Literatur vorgeschlagenen Programmiersprachen für Parallelrechner und stellen Sprachmittel vor, die einerseits hochsprachliche und problemorientierte Software-Entwicklung für Parallelrechner ermöglichen und andererseits für unterschiedlichste parallele Architekturformen in effizienten Code übersetzbar sind. Diese Sprachmittel sind von uns in der Sprache Modula-2* realisiert, die eine Erweiterung von Modula-2 darstellt. Andere bekannte Hochsprachen könnten analog erweitert werden.

2. Parallele Hardware. Wir beschreiben die Nachteile der gegenwärtig verfügbaren Parallelrechnerarchitekturen, die deren Programmierbarkeit einschränken. Ferner stellen wir mit Triton/1 neuartige

Architekturansätze vor, die die Übersetzung paralleler Sprachen wie Modula-2* unterstützen, d.h. dem Programmierer konzeptuell die Verwendung eines abstrakten Maschinenmodells gestatten.

2 Modula-2*

Am Beispiel von Modula-2* stellen wir Sprachmittel vor, die erforderlich sind, um gute Programmierbarkeit zu erreichen. Algorithmen können in Modula-2* leicht und maschinenunabhängig formuliert werden; durch ihre Lesbarkeit sind Programme einfach zu warten.

Die meisten der bereits eingeführten Programmiersprachen für parallele und hochparallele Maschinen, wie etwa *LISP, C*, MPL, VAL, SISAL, Occam, Ada, Fortran90, FortranD, BLAZE, DINO und Kali [19, 6, 25, 21, 20, 26, 1, 9, 24, 10, 23, 28, 17] haben zumindest einen der folgenden Nachteile:

- **Manuelle Virtualisierung.** Die Abbildung des gewünschten Parallelismus, der durch die oftmals variierende Problemgröße bestimmt wird, auf die tatsächlich vorhandene Prozessoranzahl, die in der Regel fest ist, muß explizit programmiert werden. Dies ist nicht nur eine ermüdende, sich ständig wiederholende Aufgabe, sondern macht die Programme auch in hohem Maße unportabel.
- **Manuelle Daten-Allokation.** Neben dem Algorithmus muß bei Maschinen mit verteiltem Speicher die Zuordnung von Datenelementen zu Speichermodulen von Hand spezifiziert werden. Durch die enge Kopplung von Algorithmenentwurf und Datenzuordnung einerseits und durch die eingeführte Topologieabhängigkeit andererseits werden die erstellten Programme wenig lesbar und nur für einen Maschinentyp verwendbar.
- **Manuelle Kommunikation.** Oft wird vom Programmierer verlangt, die Interprozeßkommunikation in Form von *send*- bzw. *get*-Operationen selbst zu programmieren. Dieser Ansatz führt zu schwer lesbaren und kaum überprüfbaren Programmen, insbesondere dann, wenn durch Asynchronität Indeterminismus entsteht. Die Topologie des Kommunikationsnetzes beeinflußt den Algorithmenentwurf, z.B. wenn aus Effizienzgründen versucht werden muß, ein schnelles Nachbarschaftsnetz auszunutzen, und führt zu Unportabilität, wenn eine gewünschte Topologie auf die tatsächlich vorhandene Netzstruktur explizit abgebildet werden soll.
- **Festlegung entweder auf SIMD oder auf MIMD.** Oben erwähnte parallele Programmiersprachen spiegeln immer Eigenschaften genau der Architekturformen wieder, für die sie entworfen wurden. Auf SIMD-Maschinen ist der Programmierer an strikte Synchronität gebunden, auch wenn das gegebene Problem dies nicht erfordert. MIMD-Maschinen erfordern komplizierte manuelle Synchronisation, da dazu keine Hardware-Unterstützung vorhanden ist. Da nicht das zu lösende Problem den Lösungsansatz bestimmt, sondern spezielle Maschinendetails, sind Portabilität der Programme und Einsatzspektrum der Sprachen deutlich eingeschränkt.

In Modula-2* wurde ein Großteil dieser Probleme gelöst. Die Sprache abstrahiert von der Speicherorganisation und von der Anzahl real vorhandener Prozessoren. Die Verteilung der Daten auf die Prozessoren wird automatisch vorgenommen, wobei der Programmierer optional Hinweise für den Übersetzer angeben kann. Ein Kommunikationsprotokoll ist nicht sichtbar. Stattdessen werden Lese- und Schreibzugriffe auf einen (virtuell) gemeinsamen Adreßraum automatisch auf Kommunikationsprimitive der Architektur abgebildet. Ein gemeinsamer Speicher ist jedoch nicht erforderlich. Der Parallelismus ist in Modula-2* explizit; der Programmierer kann zwischen synchroner und asynchroner Ausführung bei jeder Granularität wählen. Insbesondere kann der Programmierer also einen SIMD-Modus anwenden, wenn häufige oder schwierige Synchronisationen erforderlich sind, oder einen MIMD-Modus benutzen, wenn SIMD eine Unterauslastung des Rechners mit sich bringen würde. Beide Modi können beliebig verschachtelt werden.

Der datenparallele Ansatz, der in [14] vorgestellt und in Sprachen wie *LISP, C* und MPL realisiert wurde, hat die Maschinenabhängigkeit paralleler Programme deutlich reduziert und erfreut sich einer wachsenden Beliebtheit. Datenparallelismus erweitert das synchrone SIMD-Modell mit einem globalen Namensraum, der die Notwendigkeit expliziter Kommunikationsprimitiva in der Sprache hinfällig macht. Ferner erlaubt Datenparallelismus, die Anzahl der (virtuellen) Verarbeitungseinheiten als eine Funktion der Problemgröße zu betrachten, statt diese von der Architektur der Maschine abhängig zu machen.

Der datenparallele Ansatz hat drei wichtige Vorteile: (1) Er ist eine natürliche Erweiterung des sequentiellen Programmierens. Die einzige parallele Instruktion, eine synchrone **forall** Anweisung, ist eine simple Erweiterung der wohlbekannten **for** Anweisung und daher leicht verständlich. (2) Die Fehlersuche in datenparallelen Programmen ist nur unwesentlich schwieriger als diejenige in sequentiellen Programmen. Der Grund dafür ist, daß lediglich ein einziger Programmablauf zu betrachten ist, während bei reinen MIMD-Ansätzen die vielen parallelen Prozesse mit unterschiedlichen relativen Geschwindigkeiten ablaufen, was zu einer kombinatorischen Explosion der Anzahl möglicher Programmzustände führt. (3) Ferner gibt es eine große Anzahl von datenparallelen Algorithmen; die meisten in Lehrbüchern, wie etwa [2, 12], vorgestellten Algorithmen basieren auf dem datenparallelen Ansatz. Fox [11] analysierte 84 Algorithmen und konnte mehr als 80% leicht in datenparallele Algorithmen umformen. Ferner sind alle systolischen und Vektor-Algorithmen Sonderfälle von datenparallelen Algorithmen.

Aber Datenparallelismus hat, zumindest in der Form in der er heute realisiert ist, auch deutliche Nachteile, die geradlinige Software-Erstellung erheblich erschweren: (1) Datenparallelismus basiert auf einem synchronen Ausführungsmodell. Auch wenn für ein Problem keine gute synchrone Lösung gefunden werden kann, gibt es keine Möglichkeit, aus dieser Synchronität zu entfliehen. (2) Es gibt keinen geschachtelten Parallelismus. Ist also einmal eine parallele Aktivität gestartet, können die Prozesse keine weiteren parallelen Aktivitäten mehr auslösen. Diese Eigenschaft beschränkt das Einsatzspektrum datenparalleler Ansätze erheblich; z.B. ist parallele Suche in irregulären Suchräumen schwierig zu implementieren. Das Ergebnis dieser Einschränkung ist, daß die entstehenden Programme strikt bimodal sind: sie pendeln zwischen einem sequentiellen und einem parallelen Modus, wobei der Grad des Parallelismus festliegt, wenn ein paralleler Abschnitt betreten wird. Um den Grad des Parallelismus zu verändern, muß der Programmierer jegliche parallele Aktivität beenden und zunächst zur sequentiellen Betriebsweise zurückkehren. (3) Prozeduren können im datenparallelen Modell nur sehr eingeschränkt zur top-down-Strukturierung eines parallelen Programms verwendet werden, weil es nicht möglich ist, Prozeduren aus einem parallelen Kontext heraus aufzurufen, wenn diese wieder parallele Operationen ausführen sollen (Konsequenz aus (2)). Prozeduren können keine lokalen Daten allokieren und darauf parallel operieren, außer bei Aufruf aus einem sequentiellen Kontext heraus. Damit sind Prozeduren als Stukturierungsmittel nur etwa halb so oft einsetzbar, wie es wünschenswert wäre.

Modula-2* (vollständige Beschreibung in [31]) erhält die wesentlichen Vorteile des datenparallelen Ansatzes und behebt gleichzeitig die oben genannten Nachteile. Die im folgenden vorgestellten Sprachmittel erlauben, portable parallele Software von hoher Qualität und langer Lebensdauer zu erstellen ohne auf Effizienz verzichten zu müssen. Die folgende Liste gibt einen kurzen Überblick über die wichtigsten Vorteile:

- Modula-2* stellt eine echte Obermenge des datenparallelen Ansatzes dar und erlaubt sowohl synchrone als auch asynchrone parallele Programme.
- Modula-2* ist problemorientiert, d.h. maschinenunabhängig. Der Programmierer kann die Granularität des Parallelismus und den Ausführungsmodus (synchron/asynchron) beliebig auswählen und mischen.
- Parallelismus kann beliebig geschachtelt werden.
- Prozeduren können vom sequentiellen und parallelen Kontext aus aufgerufen werden und können selbst parallele Operationen enthalten.
- Die meisten Modula-2*-Programme sind sowohl für SIMD- als auch für MIMD-Maschinen in effizienten Code übersetzbar. Einen Überblick über Optimierungsansätze gibt Abschnitt 2.5.

2.1 Überblick über die Spracherweiterungen

Die notwendigen Erweiterungen an Modula-2 waren erstaunlich gering. Sie bestanden im wesentlichen aus einer synchronen und einer asynchronen Version einer **forall**-Anweisung und einigen einfachen Deklarationen, die in einer maschinenunabhängigen Weise dem Übersetzer Hinweise geben, wie verteilte Daten benutzt werden. In der Sprache ist kein Verbindungsnetzwerk sichtbar. Wir gehen von einem einheitlichen Adreßraum, jedoch nicht notwendigerweise von einem zentralen Speicher aus. Es gibt keine

expliziten Anweisungen zur Kommunikation. Stattdessen obliegt es dem Übersetzer, Lese- und Schreibzugriffe auf Speicherstellen des einheitlichen Adreßraums auf Kommunikationsoperationen abzubilden, falls die Speicherstellen nicht lokal zum benutzenden Prozeß liegen.

Die **forall**-Anweisung erzeugt konzeptuell eine Menge von parallel arbeitenden Prozessen. In der asynchronen Form operieren diese Prozesse nebenläufig und werden erst am Ende der gesamten Anweisung wieder zusammengeführt. Die asynchrone **forall**-Anweisung ist beendet, wenn alle erzeugten Prozesse ihrerseits terminiert sind. In der synchronen Variante operieren die erzeugten Prozesse „im Gleichschritt", bis sie auf einen Verzweigungspunkt, wie etwa eine **if**- oder **case**-Anweisung treffen. An solchen Verzweigungspunkten wird die Menge der synchron ablaufenden Prozesse je nach dem betretenen Ast in verschiedene Gruppen geteilt. Nur innerhalb dieser Gruppen wird dann synchron weitergearbeitet. Eine Verzweigungsanweisung ist vollständig abgearbeitet, wenn alle Gruppen ihre Zweigbearbeitung beendet haben. Dann werden die Prozeßgruppen vereinigt und führen gemeinsam die nachfolgende Anweisung wiederum synchron aus.

Varianten sowohl des synchronen als auch des asynchronen **forall** sind bereits in der Literatur vorgeschlagen. Jedoch enthält keiner dieser Vorschläge *beide* Formen, obwohl beide notwendig sind, um lesbare und portable parallele Programme schreiben zu können, die hinreichend effizient übersetzt werden können. Die synchrone Form ist oftmals einfacher zu handhaben als die asynchrone Form, da Laufzeitprobleme vermieden werden. Andererseits ist Synchronität an vielen Stellen eine zu große Beschränkung und führt leicht zu einer schlechten Maschinenauslastung. Die Kombination von synchroner und asynchroner Form in Modula-2* erlaubt es hingegen, das volle Spektrum paralleler Programmierung zu erschließen. Die **forall**-Anweisung hat folgende Syntax:

ForallStatement = FORALL ident ":" SimpleType IN (PARALLEL | SYNC)
StatementSequence
END.

Der durch die **forall**-Anweisung eingeführte Identifikator ist lokal zu dieser Anweisung und dient als Laufzeitkonstante. *SimpleType* bezeichnet eine Aufzählung oder einen (Unter-)Bereich. Das **forall** erzeugt so viele Prozesse, wie *SimpleType* Elemente besitzt und weist der Laufzeitkonstanten eines jeden dieser Prozesse einen anderen Wert dieses Typs zu. Anschließend führen die erzeugten Prozesse die Anweisungsfolge *StatementSequence* aus.

In den folgenden Abschnitten stellen wir die Semantik der asynchronen und der synchronen Variante der **forall**-Anweisung vor. Dabei wird deutlich, daß beide Formen und die vom Programmierer frei wählbare Granularität erforderlich sind, um das ganze Spektrum möglicher Algorithmen leicht lesbar und ohne unnötige Komplexität formulieren zu können.

2.2 Das asynchrone forall

Die von einem **asynchronen forall** erzeugten Prozesse führen die *StatementSequence* nebenläufig und ohne irgendeine implizite zwischenzeitliche Synchronisation aus. Die Ausführung des **forall** terminiert, wenn alle erzeugten Prozesse beendet sind. Das asynchrone **forall** beinhaltet daher genau einen Synchronisierungspunkt, nämlich am Ende der gesamten Anweisung. Jede weitere Synchronisation muß explizit, d.h. mit Hilfe von Semaphoren und den Operationen *SIGNAL* und *WAIT* implementiert werden.

Im folgenden Beispiel wird mit Hilfe des asynchronen **forall**s die Addition zweier Vektoren implementiert:

```
FORALL i:[0..N-1] IN PARALLEL
  z[i] := x[i] + y[i]
END
```

Da im Beispiel niemals zwei der erzeugten Prozesse das selbe Datenelement benutzen, ist keine zeitliche Ordnung der Operationen auf den Komponenten erforderlich. Die N Prozesse dürfen mit beliebiger Geschwindigkeit ablaufen. Das **forall** terminiert, sobald alle komponentenweisen Additionen ausgeführt sind.

Ein komplizierteres Beispiel illustriere das rekursive Erzeugen von Prozessen. Die Prozedur *ParSearch* durchsucht parallel einen gerichteten, möglicherweise zyklischen Graphen an der Wurzel beginnend und ist daher mit einer parallel ausgeführten Tiefensuche am besten zu vergleichen. Da das asynchrone **forall** verwendet wird, bleibt offen, in welcher Reihenfolge die Knoten des Graphen besucht werden.

```
PROCEDURE ParSearch( v: NodePtr );
BEGIN
  IF Marked( v ) THEN RETURN END;
  FORALL s:[0..v^.successors-1] IN PARALLEL
    ParSearch( succ(v, s) )
  END;
  visit( v );
END ParSearch;
```

Dabei erzeugt *ParSearch* jeweils so viele Prozesse, wie der gerade besuchte Knoten Nachfolger hat. Ehe ein neuer Knoten besucht wird, muß festgestellt werden, ob dieser Knoten bereits besucht und markiert worden ist. Da mehrere Prozesse einen Knoten gleichzeitig erreichen könnten, findet dieser Test und die Markierung in einem kritischen Abschnitt statt. Die notwendigen Operationen und die Verwaltung der Semaphore ist in der Prozedur *Marked* implementiert. Ist der Graph zyklenfrei, d.h. ein Baum, so ist keine Markierung erforderlich.

Das Beispiel zeigt, daß viele Algorithmen erst leicht formulierbar werden, wenn asynchrone Parallelität und die uneingeschränkte Schachtelungsmöglichkeit von Parallelität in der Sprache verfügbar sind. Der Leser möge zum Vergleich versuchen, die rekursive Graphensuche etwa in Fortran90, FortranD oder C* zu formulieren.

2.3 Das synchrone forall

Im Gegensatz zum asynchronen **forall** führen die aktiven Prozesse eines **synchronen forall** jede einzelne Anweisung der *StatementSequence* „im Gleichschritt“ aus. Um diese Betriebsweise zu verdeutlichen, sei hier die Semantik für einige Anweisungen angegeben:

- Ist eine einzelne Anweisung selbst wieder aus mehreren Anweisungen aufgebaut (z.B. zusammengesetzte Anweisungen oder Prozeduraufrufe), dann werden dieser Anweisungen im Gleichschritt nacheinander ausgeführt.
- Bei verzweigenden Anweisungen, wie `IF C THEN SS1 ELSE SS2 END`, wird die Menge der teilnehmenden Prozesse in *disjunkte* und *unabhängig voneinander operierende* Teilmengen aufgeteilt. Jede dieser Teilmengen führt den ihr zugeordneten Programmzweig, also `SS1` oder `SS2`, synchron aus. Die Ausführung der gesamten Anweisung ist beendet, wenn die Prozesse aller Teilmengen fertig sind.
- Bei Iterationsanweisungen, wie etwa `WHILE C DO SS END`, wird die Menge der teilnehmenden Prozesse bei jeder Iteration in zwei disjunkte Teilmengen aufgeteilt, nämlich die der *aktiven* und die der *inaktiven* Prozesse (bezogen auf diese Anweisung). Zu Beginn sind alle Prozesse aktiv, die die Schleife betreten. Jede Iteration startet mit der synchronen Auswertung der Schleifenbedingung `C`. Alle Prozesse, die `C` zu `FALSE` auswerten, werden inaktiv. Die restlichen Prozesse bilden die Menge der aktiven Prozesse, die die Anweisungsfolge `SS` unisono ausführen. Anschließend beginnt die nächste Iteration. Die Schleife terminiert, wenn die Menge der aktiven Prozesse leer wird.

Synchroner Parallelismus ähnelt stark der gleichschrittartigen Ausführung, die von SIMD-Maschinen bekannt ist, mit einem wichtigen Unterschied bei der parallelen Ausführung von Programmzweigen.

Als Beispiel diene hier die parallele Berechnung aller Postfix-Summen eines Vektors V der Länge N. Durch das Programm wird $V[i]$ durch die Summe aller Vektorelemente $V[i] \ldots V[N-1]$ ersetzt. Durch eine rekursive Verdopplung, wie sie etwa in [14] beschrieben ist, können diese Summen mit einer Zeitkomplexität von $O(\log N)$ berechnet werden.

```
VAR V : ARRAY[0 .. N-1] OF REAL;
VAR s : CARDINAL;
BEGIN
   s := 1;
   WHILE s < N DO
      FORALL i:[0..N-1] IN SYNC
         IF (i+s)<N THEN
            V[i]:= V[i]+V[i+s]
         END
      END;
      s := s * 2
   END
END
```

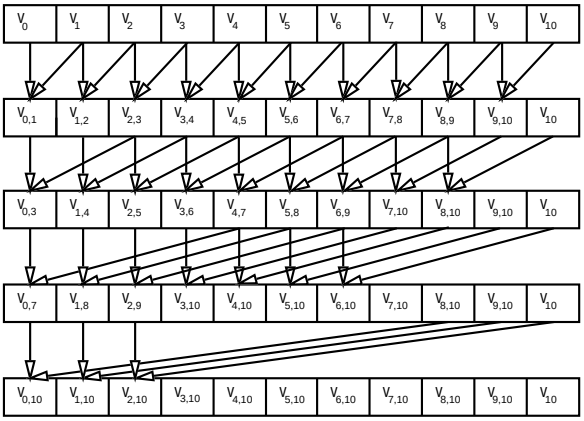

Abbildung 1: Berechnung der Postfix-Summen eines Vektors

Die Abbildung illustriert den Ablauf. Der Algorithmus basiert auf der Berechnung partieller Summen der Länge $s = 2^j$, wobei j die Anzahl der Iterationen angibt. Das innere **forall** erzeugt N Prozesse. Beachte, daß hier eine eins-zu-eins-Abbildung zwischen Prozeß und Datenelement vorliegt. In jeder Iteration wird die Länge der partiellen Summen verdoppelt, indem parallel jeweils benachbarte partielle Summen addiert werden. Die **if**-Anweisung im Inneren des **forall** schaltet dabei diejenigen Prozesse ab, die nicht mehr an der Berechnung teilnehmen dürfen.

Auch hier wird durch das Beispiel verdeutlicht, welchen Einfluß die implizite Kommunikation, der gemeinsame Adreßraum und die gegebene Synchronität auf die Lesbarkeit des Algorithmus haben. Der Aufruf an den Leser sei angeschlossen, dieses Beispiel in einer der bekannten Sprachen mit expliziter Kommunikation, etwa in Occam, zum Vergleich zu implementieren.

2.4 Allokation von Feldern

Die Zuordnung von Daten und Prozessen zu Prozessorelementen ist wichtig für Maschinen, die keine einheitliche Zugriffszeit für alle Speicherstellen haben. Schlechte Zuordnungen können viele Kommunikationsoperationen zur Folge haben.

Die meisten der erwähnten Programmiersprachen erfordern vom Programmierer gleichzeitig Entscheidungen über Algorithmus *und* Daten-Allokation. Die Allokation geschieht dabei auf unterschiedlichem Abstraktionsniveau, beginnend bei physikalischen Speichermodulen [25] über detaillierte Abbildungs- und Zuordnungsvorschriften [10] bis hinauf zur abstrakten Spezifikation der Topologie eines zugrundeliegenden Prozessornetzwerks [4, 3].

Ähnlich wie die Entwickler von Fortran90 sind wir überzeugt, daß zur Formulierung klarer Algorithmen eine explizite Daten-Allokation durch den Programmierer eher hinderlich ist und nur als Konzession gegenüber einer noch unausgereiften Übersetzertechnologie zu verstehen ist.

In Fortran90 tritt Parallelismus nur in Form elementweiser Skalaroperationen auf Feldern auf. Da diese stark eingeschränkte und hoch strukturierte Form des Parallelismus der Programmanalyse leicht zugänglich ist, können gute Allokationen automatisch gefunden werden [16]. In Modula-2* läst sich hingegen ein viel breiteres Spektrum paralleler Anweisungen formulieren, die nicht notwendig an die Benutzung von Feldern gekoppelt sind. Daher kann eine Allokation, die ausschließlich auf einer Programmanalyse beruht, nicht immer optimal sein.

Um den Übersetzer mit Zusatzinformationen zu unterstützen, die einerseits die beabsichtigte Verwendung der Felder widerspiegelt und andererseits von Maschinendetails abstrahiert, damit ein hohes Maß an Portabilität erreicht werden kann, wird in Modula-2* die Allokation von Feldern optional durch ein Schlüsselwort pro Dimension gesteuert, das auf abstrakter Ebene die beabsichtigte Verwendung dieser Dimension im Algorithmus widerspiegelt. Der Übersetzer hat auf diese Weise zusammen mit den Analy-

seergebnissen genügend Information, um eine ausreichend gute Abbildung der Daten auf eine gegebene Maschine zu finden.

Die folgenden Pragmas erachten wir als sinnvoll, um das Spektrum zwischen Lokalität aller Operationen und vollständiger Parallelität abdecken zu können. Die modifizierte Deklarationssyntax für Felder ist im folgenden angegeben:

```
ArrayType = ARRAY SimpleType [allocator]
                  {„,„ SimpleType [allocator]} OF type.
allocator = LOCAL | SPREAD | CYCLE | RANDOM | SBLOCK | CBLOCK.
```

Array-Elemente, deren Indices nur in einer Dimension differieren, die mit `LOCAL` markiert ist, werden in demselben Prozessor abgelegt.

Dimensionen mit dem Allokator `SPREAD` werden in Segmente aufgeteilt, die ihrerseits den Prozessoren zugeordnet werden. Ein Vektor mit n Elementen wird P Prozessoren zugeordnet, indem jedem einzelnen Prozessor $\lceil n/P \rceil$ aufeinanderfolgende Elemente zugeordnet werden. Dieses Allokationsschema benutzt alle zur Verfügung stehenden Prozessoren und minimiert die Kosten von Nachbarschaftskommunikation.

Dimensionen mit dem Allokator `CYCLE` werden in einem Umlaufverfahren auf die zur Verfügung stehenden Prozessoren verteilt. Stehen P Prozessoren zur Verfügung, so werden alle Elemente, deren Indices identisch modulo P sind, einem Prozessor zugeordnet. Im Gegensatz zu `SPREAD` maximiert `CYCLE` die Kosten von Nachbarschaftskommunikation, da benachbarte Array-Elemente stets in unterschiedlichen Prozessoren liegen. Dieses Allokationsschema führt jedoch zu einer erheblich besseren Ausnutzung der Maschine, wenn ein Algorithmus auf Untersegmenten eines Vektors arbeitet, während andere Segmente unbeachtet bleiben.

Dimensionen mit dem Allokator `RANDOM` werden zufällig über die verfügbaren Prozessoren verteilt. Im Gegensatz zu `CYCLE` führt `RANDOM` zu einer besseren Prozessorauslastung, wenn ein paralleler Algorithmus auf diese Dimension mit einem zufälligen Muster zugreift.

Wenn `SPREAD`, `CYCLE` oder `RANDOM` für aufeinanderfolgende Dimensionen verwendet werden, so werden diese „ausgerollt“, d.h., es wird ein Pseudo-Vektor gebildet, dessen Länge sich aus dem Produkt der Längen der einzelnen Dimensionen ergibt. Dieses Schema führt zu einer Verteilung der Array-Daten, die im allgemeinen eine bessere Auslastung der vorhandenen Prozessoren gewährleistet, als dies bei der Anwendung von `SPREAD`, `CYCLE` oder `RANDOM` für einzelne Dimensionen der Fall wäre.

Die Allokatoren `SBLOCK` und `CBLOCK` wenden `SPREAD` und `CYCLE` auf individuelle Dimensionen an. Wird `SBLOCK` für zwei aufeinanderfolgende Dimensionen verwendet, so werden „rechteckige“ Datensegmente auf einzelne Prozessoren abgebildet. Damit wird die Nachbarschaftskommunikation in allen Dimensionen unterstützt, was natürlich insbesondere dann vorteilhaft ist, wenn die zugrundeliegende Hardware ebenfalls eine derartige Kommunikationsstruktur bereitstellt.

Durch `CBLOCK` für zwei Dimensionen werden ebenfalls „rechteckige“ Datensegmente gebildet und auf einzelne Prozessoren abgebildet. Zuvor werden jedoch die Zeilen und Spalten mit einem Umlaufverfahren vermischt. Wieder unterstützt `SBLOCK` die Nachbarschaftskommunikation, während `CBLOCK` in manchen Fällen eine bessere Maschinenauslastung erreicht.

In der Literatur bereits vorgestellte Vorschläge zur Steuerung der Verteilung von Daten auf Prozessoren unterscheiden sich deutlich von unserem Ansatz. Bei Callahan und Kennedy [7] muß der Programmierer explizite Abbildungsfunktionen von Array-Indices auf Prozessornummern angeben. Bei Modula-2* werden diese Abbildungsfunktionen automatisch erzeugt. Gleichzeitig bleibt das Programm unabhängig von der Anzahl an physikalischen Prozessoren. Der Vorschlag von Mehrotra und Rosendale [22] kommt unserem am nächsten, bietet jedoch nicht die volle Allgemeinheit und ist auch nicht maschinenunabhängig, da die Anzahl und Anordnung der Prozessoren im Quellprogramm angegeben werden muß.

2.5 Übersetzung von Modula-2*

Die eingeführten Sprachmittel lassen sich bereits für die heute verfügbaren Parallelrechner gut übersetzen. Dies wiesen wir für SIMD- und MIMD-Architekturen nach, indem für die Connection Machine ein Modula-2*-Übersetzer erstellt worden ist [18] und für MIMD ein Verfahren angegeben worden ist, mit dessen Hilfe synchrone **forall**-Anweisungen in asynchrone umgeformt werden können [13], die dann direkt auf

MIMD-Rechnern implementiert werden können. Die Grundlagen unserer Transformationen sind in [27] dokumentiert. Einige der auftretenden Probleme, wie z.B. das der Realisierung paralleler Zeiger, konnten jedoch auf heutigen Maschinen nur wenig effizient gelöst werden. Die notwendige Unterstützung durch die Hardware, wie z.B. durch indirekte Adressierung in einem gemeinsamen Adreßraum, ist im nächsten Abschnitt beschrieben und wird in Triton/1 implementiert.

Nachdem die Praktikabilität unseres Sprachentwurfs nachgewiesen war, gelang es uns durch Optimierungen, die erreichten Laufzeiten erheblich zu verkürzen. Manche Optimierungen können im Bereich der Übersetzung für Parallelrechner die Geschwindigkeit um eine ganze Größenordnung (oder sogar mehr) verbessern. Im Zentrum unserer Optimierungsbemühungen stehen:

Elimination von Synchronisierungspunkten. Viele durch die Semantik von Modula-2* vorgegebene Synchronisierungspunkte innerhalb eines synchronen **forall** sind in realen Programmen nicht erforderlich. Sie können also entfallen, ohne die Bedeutung des Programms zu verändern. Um solche Fälle zu erkennen, kommen Techniken zur Anwendung, die ähnlich in parallelisierenden Fortran-Übersetzern verwendet werden, um Parallelität in **for**-Schleifen zu erkennen.

Durch die Elimination der Synchronisierungspunkte gelingt es, synchrone **forall**-Anweisungen mit hoher Effizienz auf MIMD-Rechnern zu implementieren. Je weiter wir die Datenabhängigkeitsanalyse verfeinern, desto weniger Synchronisierungspunkte sind notwendig und desto weniger Kommunikationsaufwand ist auf MIMD-Rechnern zur Synchronisation erforderlich. Die Effizienz des entwickelten Transformationsverfahrens [13], das ohne die Sprachebene von Modula-2* zu verlassen, synchrone **forall**-Anweisungen in eine Menge von asynchronen **forall**-Anweisungen umsetzt, verbessert sich also mit der Güte der Datenabhängigkeitsanalyse.

Doch die Elimination bringt auch für SIMD-Rechner erhebliche Vorteile. Unser Übersetzer für die Connection Machine profitierte von der automatischen Virtualisierung, die in C/Paris angeboten wird. Automatische Virtualisierung bedingt aber sehr pessimistische Annahmen über eventuell bestehende Datenabhängigkeiten und muß daher intensiv von Hilfsvariablen Gebrauch machen; allein deren Anzahl macht die Benutzung von Registern unmöglich, wie das folgende Beispiel zeigt.

```
FORALL i: [0..N-1] IN SYNC
    A[i] := (A[i] + 1) / 2
END
```

Die beiden folgenden Programmstücke geben zwei Möglichkeiten an, wie eine Virtualisierung von N konzeptuellen Prozessen auf p realen Prozessoren vorgenommen werden kann. Wir benutzen dabei zur Vereinfachung der Darstellung wieder Modula-2*, obwohl die Virtualisierung vom Übersetzer auf einem niedrigerem Niveau realisiert wird.

```
s := CEILING(N, p)
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1 DO
        TMP[i] := A[i] + 1;
        TMP[i] := TMP[i] / 2
    END
END;
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1 DO
        A[i]   := TMP[i]
    END
END
```

```
s := CEILING(N, p)
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1 DO
        reg := A[i];
        reg := reg + 1;
        reg := reg / 2;
        A[i]:= reg
    END
END
```

Das Programmfragment links wird von einer automatischen Virtualisierung etwa auf der Connection Machine benutzt. Während links ein temporäres Array benötigt wird, kann ein Übersetzer die Abwesenheit von Datenabhängigkeiten erkennen und ein Register verwenden, wie auf der rechten Seite angegeben. Damit wird nicht nur Speicherplatz eingespart, sondern auch die Anzahl der Hauptspeicherzugriffe auf ein Drittel reduziert. Zusätzlich können eventuell vorhandene Pipeline-Eigenschaften der Prozessoren ausgenutzt werden. Durch die Elimination von Synchronpunkten wird nicht nur eine effiziente Übersetzung für MIMD sondern auch eine effiziente Virtualisierung auf SIMD-Rechnern erst möglich.

Zweigvereinigung. Es muß nicht nur möglich sein, synchrone **forall**-Anweisungen effizient auf einer MIMD-Maschine zu realisieren, wozu wir Synchronpunktelimination verwenden, sondern auch die asynchronen **forall**-Anweisungen müssen von einer SIMD-Maschine effizient abgearbeitet werden, ohne daß ein Großteil der Maschine ungenutzt bleibt.

Wir arbeiten gegenwärtig an einer „Zweigvereinigung“: Obwohl die Semantik von Modula-2* vorschreibt, daß an Verzweigungspunkten und in asynchronen **forall**-Anweisungen gegenseitig unabhängige Gruppen von Prozessen gebildet werden, ist es natürlich erlaubt, diese Gruppen temporär zu vereinigen, falls sie tatsächlich den gleichen Code benutzen. Gleiche Segmente in verschiedenen Programmzweigen, die eine solche temporäre Vereinigung zulassen, können mit Algorithmen gefunden werden, die zur Bestimmung der *Longest Common Subsequence* [29] angewendet werden.

Zwischensprache. Für alle uns verfügbaren Parallelrechner und zukünftig auch für Triton/1 steht eine Programmierschnittstelle zur Verfügung, die als Obermenge von C betrachtet werden kann. Deshalb erzeugen wir momentan direkt aus dem attributierten Strukturbaum heraus den geforderten C-Code. Im Laufe der zweiten Phase arbeiten wir an einer Zwischensprache, die es ermöglichen soll, für SIMD- und MIMD-Rechner Maschinencode zu erzeugen. Vor der Erzeugung der Zwischensprache steht natürlich die vollständige Transformation des vom Programmierer gewünschten Parallelismus auf das jeweils zur Verfügung stehende Parallelitätskonzept der unterliegenden Hardware.

3 Triton/1 Prototyp

Die schlechte Programmierbarkeit heutiger Parallelrechner läßt sich vor dem Hintergrund ihrer Entwicklungsgeschichte verstehen: sie wurden vorwiegend basierend auf Machbarkeitsüberlegungen der Hardware-Entwickler konzipiert; Programmierbarkeit war dabei kein Entwurfsziel. Erst nach Fertigstellung der Prototypen wurden Sprachen ergänzt, die jedoch stark durch die jeweilige Architektur geprägt sind.

Wir zeigen die wesentlichen Nachteile bezüglich der Programmierbarkeit der verfügbaren Hardware-Entwürfe auf und entwickeln einige Alternativen dazu, die wir in einem von uns entworfenen Prototypen exemplarisch realisieren. Triton/1 vereinigt die Vorteile von typischen SIMD- und MIMD-Maschinen in einer SAMD-Architektur [1] und berücksichtigt durch ein neuartiges, leistungsfähiges Kommunikationsnetzwerk die konzeptuellen Erfordernisse von Programmiersprachen wie Modula-2*. Triton/1 basiert damit nicht nur auf Machbarkeitsüberlegungen sondern ist stark geprägt von Forderungen der Programmiersprachen und Übersetzerbauer. Mit diesem Experiment zeigen wir Entwurfsparadigmen zukünftiger paralleler Rechner auf, die zur Programmierbarkeit und Portabilität der Software beitragen.

3.1 Schwachstellen existierender, massiv paralleler Rechner

Die Hauptschwächen der verfügbaren Parallelrechner liegen auf folgenden Gebieten:

- **Kommunikationsnetzwerk.**

  Das schwerwiegendste Manko heutiger massiv paralleler Rechner ist die zu niedrige Geschwindigkeit des Kommunikationsnetzwerks verglichen mit der vorhandenen Rechenleistung. Dieses Problem tritt bei MIMD-Rechnern meist in Form einer hohen Latenzzeit in Erscheinung: etwa 100 arithmetische Operationen oder mehr haben den gleichen Zeitbedarf wie die Versendung eines einzigen kleinen Paketes. Daher muß heute bei der Programmierung von MIMD-Rechnern die Reduktion der Nachrichtenanzahl durch Umorganisation des Algorithmus oder durch geschickte Packungsverfahren im Vordergrung stehen, wenn vernünftige Leistungen erzielt werden sollen.

  Bei SIMD-Rechnern ist das Verhältnis zwischen der Ausführungszeit einer Operation und dem Versenden eines Paketes etwas besser: etwa 1:10 für Nachbarschaftskommunikation und 1:40 für allgemeine Kommunikation [32]. Das ist jedoch noch immer nicht ausreichend. Der Programmierer ist gezwungen, alles nur Erdenkliche zu tun, um durch Organisation des Algorithmus eine gute Anpassung an die Netztopologie zu erreichen, damit die schnellere Nachbarschaftskommunikation ausgenutzt werden kann.

[1] SAMD = synchronous or asynchronous multiple data

Im Rahmen der Studien, die vor dem Entwurf von Triton/1 vorgenommen worden sind, stellte sich heraus, daß das seit langem bekannte Netzwerk von De Bruin wegen seiner Vielzahl theoretisch höchst akzeptabler Eigenschaften, seiner Kostengünstigkeit und seiner Praktikabilität ein geeignetes Kommunikationsnetzwerk ist, mit dem obige Nachteile vermieden werden. Dieses Netzwerk wurde bisher – wie wir zeigen werden zu Unrecht – verworfen, da keine Verklemmungslosigkeit erreicht werden konnte. Wir stellen im Abschnitt 3.4 ein Verfahren vor, mit dem Verklemmungslosigkeit erreicht wird.

- **Synchronität versus Asynchronität.**

  Bereits in der Präsentation von Modula-2* stellten wir heraus, daß sowohl strikt synchrone Ausführung als auch Asynchronität in der Programmiersprache erforderlich sind, um hohe Software-Qualität zu erreichen.

  Heutige Parallelrechner unterstützen aber stets nur eine dieser Betriebsweisen. SIMD-Maschinen bieten Synchronität an, jedoch können unterschiedliche Äste der Programme nicht nebenläufig abgearbeitet werden. Oft bleiben viele der Prozessorelemente zeitweilig ungenutzt. MIMD-Maschinen, die heute meist mit der Methode SPMD[2] programmiert werden, können zwar unterschiedliche Programmäste nebenläufig abarbeiten, erzwingen aber häufige teure Synchronisation und vergeuden durch redundante Programmspeicherung viel Hauptspeicher.

  In Triton/1 erlauben wir das Nebeneinander von SIMD- und MIMD-artigem Parallelismus. Wir stellen eine Architektur vor, die sowohl Hardware-Unterstützung zur schnellen Synchronisierung bietet als auch einen asynchronen Betrieb zuläßt, wo dies effizienzsteigernd ausgenutzt werden kann.

- **Skalierbarkeit und Ausgewogenheit.**

  Betrachtet man die bekannten Architekturen für massiv parallele Rechner genauer, so stellt man fest, daß diese schlecht skalierbar sind und daß deren Funktionalität oftmals nur ungenügend ausgewogen ist. So erfordert die Hyperkubus-Topologie bei einer Verdopplung der Prozessoranzahl für jeden Prozessor eine zusätzliche Verbindungsleitung. Diesem Vorgehen stehen sowohl beim Chip-Entwurf als auch bei der Verdrahtung sehr viele Hindernisse im Wege. Die Gitterarchitektur skaliert auf dem Papier mühelos, doch ist die durch den ungünstigen Durchmesser ($\sqrt{n}$) bedingte Laufzeit der Nachrichten ein ernstzunehmendes Problem. Neben der reinen Skalierbarkeit der Verknüpfungsstruktur in Architektur und Realisierbarkeit müssen auch die Leistungsfähigkeiten der verschiedenen Komponenten des Entwurfs aufeinander abgestimmt werden. Obwohl diese Abstimmung bei der Konzeption sequentieller Rechner als notwendig erachtet wird, wurde sie für Hochparallelrechner bisher eher vernachlässigt. Es ist unklar, welche Leistung etwa das Vermittlungsnetzwerk bei Vorgabe einer Prozessorleistung haben sollte, damit keine Engpässe entstehen.

  Während (MIMD-)Hersteller vorwiegend darauf setzen, den Prozessor „hochzuzüchten“, erreichen wir in Triton/1, mit einem auf die Prozessorleistung abgestimmten Netzwerk eine Ausgewogenheit von Kommunikations- und Rechenleistung für feinkörnige parallele Algorithmen.

  Die Ein-/Ausgabeleistung ist bereits bei sequentiellen Rechnern ein Engpaß. Dieser Engpaß gewinnt bei Parallelrechnern erheblich an Bedeutung, da durch die größere Prozessorleistung noch wesentlich mehr Daten zu bearbeiten sind. In Triton/1 ist daher eine massiv parallele Ein-/Ausgabe integriert, die nicht nur mit der Rechnergröße skaliert, sondern deren Durchsatz sich proportional zur Rechnergröße verhält. Durch die Integration vieler Platteneinheiten wird es sinnvoll, über neuartige Dateikonzepte und deren Repräsentation für den Benutzer nachzudenken und Betriebssystemaufgaben wie etwa virtuelle Speichertechnik, parallele Seitenersetzungsstrategien und darauf aufbauend einen echten Mehrbenutzerbetrieb anzugehen. Dies ist notwendig, um festzustellen, ob Parallelrechner als künftige Allzweck-Architekturen breite Verwendung finden können.

- **Übersetzerunterstützung.**

  Während der Arbeit am Modula-2*-Übersetzer kristallisierten sich eine Reihe von notwendigen Eigenschaften heraus, die Hochparallelrechner erfüllen müssen, damit hochsprachliche Konstrukte auf effizienten Code abgebildet werden können.

[2]SPMD = Single Program Multiple Data

Die wichtigste Forderung ist die nach einem Kommunikationsnetz, das in der Lage ist, *beliebige* Kommunikationsanforderungen effizient zu bearbeiten. Wir stellten fest, daß in vielen Fällen der Übersetzer nicht in der Lage war, zwischen verschiedenen durch die Hardware angebotenen Kommunikationsalternativen zu wählen. Oft war es z.B. nicht möglich zu erkennen, daß die hochoptimierte Nachbarschaftskommunikation hätte ausgenutzt werden können.

Eine zweite Forderung in diesem Zusammenhang stellt die Trennung der Kommunikationsanforderungen in separate *Send*- und *Receive*-Anweisungen dar. Auf diese Weise kann ein Übersetzer mittels Datenflußanalyse den erzeugten Code soweit optimieren, daß sich Kommunikation und Berechnung überlappen und somit die Kommunikationszeiten weitgehend verborgen werden können.

Eine eher praktische Forderung ist die nach einem großen Haupstspeicherausbau pro Prozessor. Grund dafür ist, daß man auf den verfügbaren Maschinen (z.B. MasPar oder CM-2) bei der Realisierung vieler virtueller Prozesse sehr schnell an die Grenzen der Speicherkapazität stößt.

Weiterhin ist ein gemeinsamer Adreßraum erforderlich. Ein Prozessor sollte in der Lage sein, eine Adresse zu erzeugen, die eine eindeutig bestimmte Stelle im gesamten Speicher bezeichnet. Es sollte der Hardware obliegen, diese Adresse entweder in einen Zugriff auf den lokalen Speicher eines Prozessors abzubilden oder mit Hilfe von Kommunikation am speichernden Knoten auszuführen.

Ferner werden parallele Zeiger (indirekte Adressierung) und deren Verwendung bisher nur sehr ungenügend oder gar nicht unterstützt, obwohl deren Verwendung in vielen Fällen sinnvoll ist.

Durch die genannten Forderungen, die in Triton/1 bereits zum Großteil realisiert wurden, können Übersetzer erheblich besser unterstützt werden, als dies mit bekannten Hochparallelrechnern möglich ist.

Triton/1 bietet also die Hardware-Unterstützung, die erforderlich ist, damit hochsprachliches und portables Programmieren in der präsentierten Form erleichtert wird und zeigt gleichzeitig neuartige Architekturprinzipien auf, die in den folgenden Abschnitten detaillierter beschrieben werden.

3.2 Aufbau von Triton/1

Um Kosten und Entwicklungsaufwand möglichst klein zu halten, entwarfen wir keinen eigenen VLSI-Chip, sondern bauen die Maschine ausschließlich aus marktgängigen Teilen und programmierbaren Gate-Arrays auf. Der Triton/1 Prototyp umfaßt 260 Prozessorknoten. Die Architektur skaliert bis zu 4096 Knoten, von denen jeder aus einem Motorola MC68010 Microprozessor, einem Numerik-Coprozessor, 512 KBytes Hauptspeicher, einer Speicherverwaltungseinheit (MMU) und aus einem Netzwerkprozessor besteht. Jeder zweite Prozessorknoten ist mit einer SCSI-Schnittstelle ausgestattet, die es ermöglicht, ein Plattenlaufwerk zu betreiben. Einen Überblick gibt die Abbildung 2.

Im Prototyp Triton/1 wird die Fehlererkennung durch Paritätsprüfungen vorgenommen. Überprüft werden der Hauptspeicher, jede Netzwerkverbindung und der Massenspeicher. Fehlerhafte Bauteile, die keine Paritätsfehler erzeugen, werden durch periodische Tests der ganzen Maschine per Signaturanalyse festgestellt. Der Prototyp wird 260 Knoten mit programmierbaren Identifikationsnummern haben, von denen nur 256 an den Berechnungen teilnehmen; vier Knoten werden also als „hot-stand-by“ verwendet. An Knoten auftretende Fehler können innerhalb einer Minute neutralisiert werden, indem die Numerierung umprogrammiert und für das Netzwerk neue Nachbarschaftstabellen ausgerechnet werden.

3.3 SAMD-Architektur

In Triton/1 gelang es, die Vorteile von SIMD- und MIMD-Architekturen zu vereinigen: es ist ein schneller Wechsel zwischen synchroner und asynchroner Betriebsweise möglich. Deshalb bezeichnen wir die Architektur mit SAMD, für „synchronous or asynchronous multiple data“.

Im Bereich *hoch*paralleler Systeme scheint es kaum vorstellbar, jede der mehreren tausend Verarbeitungseinheiten sinnvoll mit *verschiedenen* Programmen zu beschäftigen. In der Tat werden die meisten heute erhältlichen hochparallelen MIMD-Rechner mit der Methode SPMD programmiert. Dabei wird *ein* Programm verwendet, welches von den verschiedenen Verarbeitungseinheiten zumindest teilweise asynchron ausgeführt wird. Dieses Verfahren erinnert stark an SIMD, der Unterschied besteht lediglich darin,

Faltblattbild

Abbildung 2: Architekturskizze von Triton/1

daß ein Programm auf einem SIMD-Rechner immer synchron ausgeführt werden muß, wodurch bei Abschaltung von Prozessoreinheiten ein Teil der theoretischen Rechenleistung verloren geht.

Um diesen Verlust quantifizieren zu können, haben wir eine Reihe von Grundalgorithmen und einige der von Fox [11] untersuchten Algorithmen in Modula-2* ausformuliert. Ferner studierten wir viele der speziell für MIMD entworfenen SPMD-Algorithmen. Wir stellten fest, daß der Verzweigungsgrad in diesen Algorithmen im Bereich kleiner Zahlen liegt und die nebenläufigen Programmäste selbst meist kurz sind und von Synchronisations- oder Kommunikationsoperationen beendet werden.

Vorteilhaft ist deshalb, wenn die Maschine schnelle Synchronisation, schnelle Kommunikation und einen effizienten Rundruf etwa von Direktoperanden und Instruktionen ermöglicht, also den wesentlichen Nachteil heutiger MIMD-Maschinen aufhebt. Gleichzeitig sollten verschiedene Programmzweige von unterschiedlichen Prozessoren selbständig und asynchron abgearbeitet werden können, was den typischerweise bemängelten SIMD-Nachteil verschwinden läßt.

In Triton/1 sind die Vorteile beider Architekturformen vereinigt: solange SIMD-Betrieb sinnvoll ist, werden Instruktionen und Direktoperanden vom Front-End an die Prozessoren geschickt; für die Bereiche eines Algorithmus, in denen ein MIMD-artiger Betrieb angezeigt ist, liegen die notwendigen Programmstücke im lokalen Speicher der Prozessorelemente. Sie kommen zur Ausführung, indem das Front-End mit dem letzten synchronen Befehl jedem Prozessor seine Startadresse mitteilt und auf MIMD-Betrieb umschaltet. Hier zahlt sich aus, daß Triton/1 auf der Basis von Universalprozessoren konstruiert ist, die eigenständig Programme interpretieren können. Am Ende dieser Programmstücke finden die Prozessoren einen speziellen Befehl zum Signalisieren an einer global-Wired-Or-Leitung, die dem Front-End das Fortsetzen des Synchronbetriebs anzeigt. Die Synchronisation im MIMD-Betrieb muß auf Software-Basis realisiert werden.

3.4 Kommunikationsnetz

Die Topologie des Netzwerkes basiert auf einem De Bruijn-Netz [5, 15], das durch die folgende Regel charakterisiert werden kann: Seien N Knoten gegeben, numeriert von 0 bis $N-1$, dann ist jeder Knoten X direkt mit denjenigen Knoten verbunden, deren Nummern gegeben sind durch $(2*X+\alpha)\mathbf{mod}N$, wobei $\alpha \in \{0..d-1\}$. Bei diesem Netzwerk entsprechen sich Ausgangsgrad und Eingangsgrad. N ist bei diesem Netzwerk nicht auf Zweierpotenzen beschränkt. Der (maximale) Abstand zweier Knoten ist $\lceil \log_d N \rceil$. Der durchschnittliche Abstand liegt deutlich unter $\log_d N$ und in Praxis nur knapp über der theoretischen unteren Grenze, also dem durchschnittlichen Abstand gerichteter Moore-Graphen. (Zum Beispiel ist der durchschnittliche Abstand eines De Bruijn-Netzes mit $N = 256$ und $d = 2$ nur 5% schlechter als der des äquivalenten Moore-Graphen, der jedoch nicht praktisch realisierbar ist.)

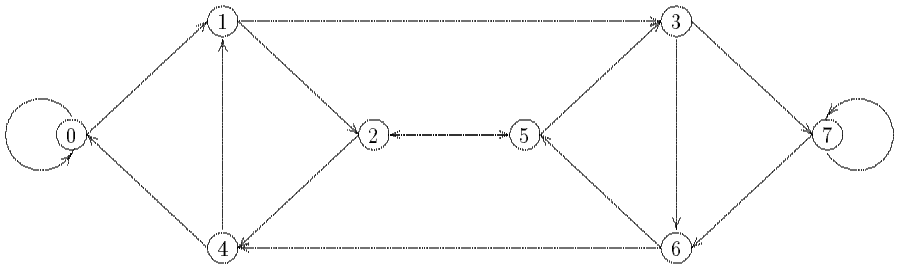

Abbildung 3: Mischungsnetzwerk für 8 Knoten

Bei unserer Implementierung benutzen wir den Grad $d = 2$, wodurch unser Netz zum Mischungsnetzwerk (perfect shuffle) wird (siehe Abbildung 3). Diese Konstruktion hat gegenüber anderen gebräuchlichen Netzwerken die Vorteile eines konstanten Verbindungsgrades pro Knoten sowie eines kleinen mittleren Durchmessers. Das Netzwerk arbeitet auf der Basis einer tabellengesteuerten Paketvermittlung. Dazu ist jeder Knoten mit einer eigenen Routing-Tabelle und drei Eingangspuffern ausgestattet. Zwei Puffer dienen der Zwischenspeicherung von Paketen, die von anderen Knoten kommen; über einen weiteren Puffer wird die Kommunikation mit dem zugeordneten Prozessor abgewickelt. Durch die Pufferung wird das Netzwerk von den Prozessoren zeitlich entkoppelt. Es wird sogar unabhängig vom lokalen Hauptspeicher der einzelnen Prozessoren. Datenpakete setzen sich zusammen aus der Adresse des Zielknotens, der Anzahl an Datenworten und den Daten selbst. Die Nutz-Paketgröße kann zwischen 1 und 506 Bytes variieren. In Abbildung 4 lassen sich die Komponenten des Netzwerkknotens wiedererkennen.

Foto + Legende

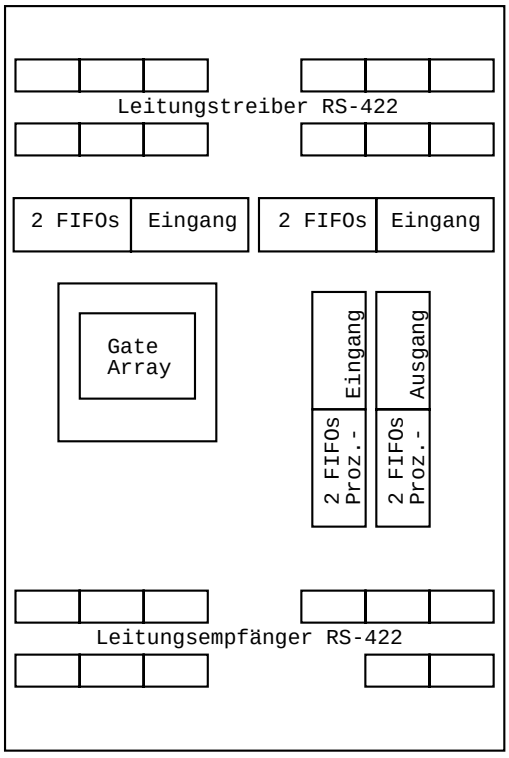

Abbildung 4: Prototypaufbau des Netzwerkknotens

Die Hauptziele beim Design des Netzwerkprozessors waren kurze Latenzzeiten, vollautomatisches, tabellengesteuertes Routing und die Verklemmungsfreiheit des Netzwerkes.

Um die geforderte kurze Latenzzeit zu erreichen, wurde der die Vermittlung bewerkstelligende Netz-

prozessor als Hardware-Bauelement auf Gatterebene entworfen und mit einem programmierbaren Gate-Array implementiert. Im vorliegenden Testaufbau erreicht der Netzprozessor eine Kommunikationsrate von 800.000 Paketen/s (32-bit Nutzdaten). Dies entpricht einer Kommunikationsleistung von 0.8 MCPS[3], die voll mit der Anzahl der Prozessoren skaliert. Das ist drei- bis fünfmal schneller als die zugehörigen Prozessoren Pakete lesen oder schreiben können. Die sich dadurch ergebenden Leistungsreserven ermöglichen auch in großen Netzen und unter großer Last des Netzwerks, die geforderten kurzen Latenzzeiten zu erreichen.

Da die Netzprozessoren vollkommen unabhängig von den eigentlichen Prozessoren arbeiten, ist es möglich, Datenübermittlungen im Netzwerk und Berechnungen durch den Prozessor parallel ablaufen zu lassen. Damit hat der Übersetzer bei vielen Anwendungen die Möglichkeit, durch entsprechende Umordnung des Maschinencodes die Kommunikations- und Rechenzeiten zu überlappen.

Um das reale Lastverhalten des geplanten Netzwerks schon im Vorfeld untersuchen zu können, wurde für das Gesamtverhalten des Netzwerks ein Simulator erstellt, der auf gemessenen Leistungsdaten des Ein-Prozessor-Prototyps basiert. Dabei wurden denkbare Betriebsmodi auf ihre Leistungsfähigkeit unter verschiedenen Lastanforderungen für den Bereich zwischen 32 und 8192 Knoten untersucht. Abbildung 5 zeigt die Ergebnisse einer Simulationsreihe, bei der wir zufällige Kommunikationsmuster untersuchten. Die einzige Voraussetzung für diese Kommunikationsmuster bestand darin, daß genausoviele Pakete wie vorhandene Prozessoren verschickt werden. Die Ergebnisse dieser Simulationen zeigen die gute Eignung dieses Netzwerks: Die Verzögerung durch das Netzwerk steigt bei proportional wachsender Knotenzahl und Last mit $O(\log n)$.

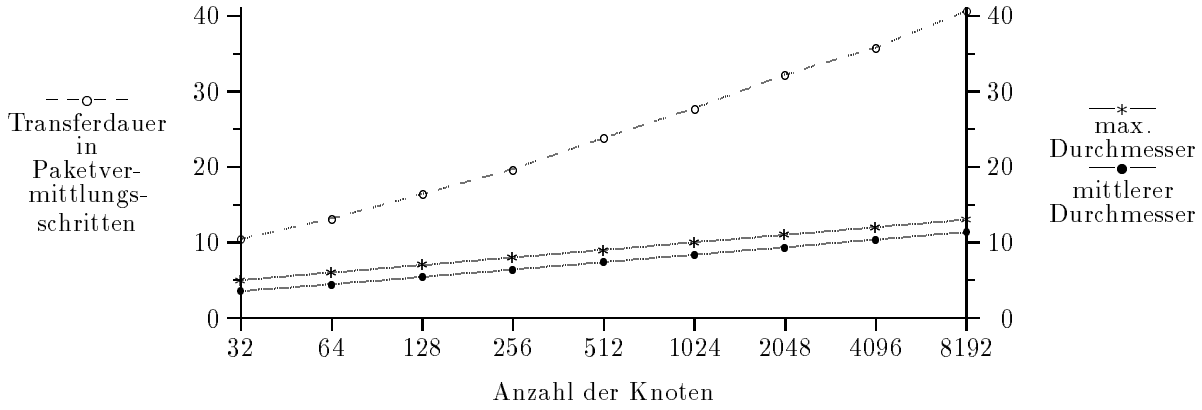

Abbildung 5: Leistungsparameter bei Zufallskommunikation

Die Robustheit gegen Überlast ist erstaunlich gut, so daß selbst wenn alle Prozessoren gleichzeitig sehr viele Pakete versenden wollen, das Netzwerk in seiner Gesamttransferleistung nicht beeinträchtigt wird. Insbesondere irreguläre Permutationen werden fast immer sehr schnell bewältigt. Alle in der Literatur als „Härtetests“ bekannten Kommunikationsmuster wie etwa die Spiegelung der Binärdarstellung der Adresse, die Transposition einer Matrix, ein Butterfly-Muster oder eine Perfect-Shuffle-Permutation werden wie jede beliebige Permutation oder sogar schneller bedient.

Beim Entwurf dieses Netzwerks wurde bewußt darauf verzichtet, einzelne regelmäßige Kommunikationsmuster gezielt zu unterstützen. Vielmehr ist das Ziel, unregelmäßige Permutationen sowie völlig zufällige Kommunikationsmuster möglichst schnell abarbeiten zu können.

Ein Problem, das dabei erst noch gelöst werden mußte, ist das der Verklemmungsgefährdung des De Bruijn-Netzes. Nach [8] ist ein Netzwerk dann verklemmungsfrei, wenn der zugehörige Abhängigkeitsgraph zyklenfrei ist. Dies läßt sich für den Hyperkubus sehr leicht beweisen; für De Bruijn-Netze gilt das nicht. Das Problem der Verklemmungsgefährdung ist in der Literatur verschiedentlich behandelt worden. Jedoch ist uns kein Verfahren aus der Literatur bekannt, welches geeignet wäre, mit den Möglichkeiten der hier verwendeten Hardware angewendet zu werden. So kann z.B. das in [8] vorgeschlagene Verfahren

[3]**M**illion **C**onnections **P**er **S**econd

in unserem Fall nicht angewendet werden, da ein Umordnen einzelner Pakete innerhalb eines Eingangspuffers nicht möglich ist.

Um dennoch Verklemmungen behandeln zu können, haben wir sich ergänzende Methoden entwickelt. Das erste Verfahren reduziert die Wahrscheinlichkeit, mit der eine Verklemmung auftritt durch eine statische Prioritätsregel. Weiterhin benutzen wir eine Zugangskontrolle, die es dem Prozessor untersagt, beliebig viele Pakete ins Netz zu schleusen und somit das Netz zu blockieren. Falls dennoch eine Stausituation auftreten sollte, verwenden wir eine Art dynamisches Umwegrouting, das in Verbindung mit der oben genannten Prioritätsregelung einen garantierten Fortschritt innerhalb des Paketvermittlungsverfahren gewährleistet. Ein ähnliches Routingverfahren wird auch im Denelcor HEP [30] verwendet, mit dem Unterschied, daß unser Verfahren ohne explizite Paketprioritäten arbeitet.

4 Bewertung

Die parallele Informatik wiederholt zur Zeit viele Entwicklungsstufen, die auch die sequentielle Informatik in den letzten vier Dekaden durchlief. Durch verbesserte Hardware-Architekturen, Betriebssysteme, Programmiersprachen und Übersetzertechnologien wird deshalb maschinenabhängiges Programmieren von Parallelrechnern bald die gleiche Bewertung erfahren, die Assemblerprogrammierung für sequentielle Rechner bereits heute hat. Neben der reinen Laufzeit rückt das Ziel der Programmierbarkeit in den Mittelpunkt.

Um dieses Ziel zu erreichen, ist es notwendig, Sprachentwurf und Hardware-Entwicklung zu koppeln und aufeinander abzustimmen, wie dies im Projekt Triton geschieht. Ausgehend von diesem Ansatz gelang es einerseits, mit Modula-2* ein hochsprachliches und portables Programmiermodell zu entwickeln, das die prinzipiellen Eigenschaften paralleler Hardware berücksichtigt, und andererseits mit Triton/1 neuartige Architekturmöglichkeiten paralleler Hardware vorzuschlagen und exemplarisch zu realisieren, die die effiziente Übersetzbarkeit der gefundenen Sprachmittel ermöglichen.

Zeitplan

Modula-2*. Ein Übersetzerprototyp für die Connection Machine ist bereits funktionsfähig. Im Frühjahr 1992 sind optimierende Übersetzer für MasPar, Transputer und Ein-Prozessor-Maschinen verfügbar. Anschließend werden weitere Optimierungen angegangen, in deren Zentrum die Verbesserung der Plazierung der Datenelemente im verteilten Speicher und das damit zusammenhängende Auswerten von Ausdrücken stehen.

Triton/1. Der Ein-Prozessor-Prototyp von Triton/1 ist im Oktober 1991 aufgebaut, die einzelnen Komponenten (Netzwerkprozessor, Prozessorplatine und Front-End-Interface) sind getestet und funktionieren spezifikationsgemäß. Wegen Testbarkeit der Prozessorplatine und einiger hier nicht angesprochener Architekturdetails werden noch Optimierungen an der Prozessorplatine durchgeführt. Anfang 1992 gehen die Platinen dann in Fertigung. Nach Bestückung der Platinen und Endaufbau wird Triton/1 bis zum Sommer 1992 fertiggestellt.

Literatur

[1] Ada. U.S. Government, Ada Joint Program Office. ANSI/MIL-Std 1815 A, Reference Manual for the Ada Programming Language, January 1983

[2] Akl, S.G.: The Design and Analysis of Parallel Algorithms. Englewood Cliffs, New Jersey: Prentice Hall 1989

[3] Barth, I., Bräunl, T., Engelhardt, S., Sembach, F.: Parallaxis version 2 user manual. Computer Science Report 2/91, Universität Stuttgart, February 1991

[4] Bräunl, T.: Massiv parallele Programmierung mit dem Parallaxis-Modell. Informatik Fachberichte Nr. 246. Berlin, Heidelberg, New York, London, Paris, Tokyo: Springer 1990

[5] De Bruijn, N.G.: A combinatorial problem. In: Proc. of the Sect. of Science Akademie van Wetenschappen, pp. 758–764, Amsterdam, June 29 1946

[6] C*. Thinking Machines Corporation, Cambridge, Massachusetts. C* Language Reference Manual, April 1991

[7] Callahan, D., Kennedy, K.: Compiling programs for distributed-memory multiprocessors. The Journal of Supercomputing, **2**, 151–169 (1988)

[8] Dally, W.J., Seitz, C.L.: Deadlock-free message routing in multiprocessor interconnection networks. IEEE Transactions on Computers, **36**(5), 547–553 (1987)

[9] Fortran90. American National Standards Institute, Inc., Washington, D.C. ANSI, Programming Language Fortran (Fortran90), Draft S8, Version 114 (X3.9-1990), January 1990

[10] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., Wu, M.-Y.: Fortran D Language Specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990

[11] Fox, G.C.: What have we learnt from using real parallel machines to solve real problems. In: Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications, Vol. 2, pp. 897–955, Pasadena, CA, February 26 – March 2 1988. New York: ACM Press

[12] Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press, 1988

[13] Heinz, E.A.: Transforming synchronous parallel FORALL contexts into equivalent asynchronous parallel FORALL contexts. Interner Bericht, Universität Karlsruhe, Fakulät für Informatik, Juli 1990

[14] Hillis, W.D., Steele, G.L.: Data parallel algorithms. Communications of the ACM, **29**(12), 1170–1183, December 1986

[15] Imase, M., Itoh, M.: Design to minimize diameter on building-block network. IEEE Transactions on Computers, **30**(6), 439–442, June 1981

[16] Knobe, K., Lukas, J.D., Steele, G.L.: Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. Journal of Parallel and Distributed Computing, **8**(2), 102–118, February 1990

[17] Koelbel, C., Mehrotra, P.: Supporting shared data structures and distributed memory architectures. In: Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 177–186, March 1990

[18] Kretzschmar, R.: Ein Modula-2* Übersetzer für die Connection Machine. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Mai 1991

[19] *Lisp. Thinking Machines Corporation, Cambridge, Massachusetts. *Lisp Reference Manual, Version 5.0, 1988

[20] McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., Thomas, R.: SISAL Language Reference Manual. Lawrence Livermore National Laboratory, March 1985

[21] McGraw, J.R.: The VAL language: Description and analysis. ACM Transactions on Programming Languages and Systems, **4**(1), 44–82, January 1982

[22] Mehrotra, P., Van Rosendale, J.: Parallel language constructs for tensor product computations on loosely coupled architectures. Technical Report 89-41, Institute for Computer Applications in Science and Engineering (ICASE), September 1989

[23] Mehrotra P., Van Rosendale, J.: The BLAZE language: A parallel language for scientific programming. Parallel Computing, **5**, 339–361, November 1987

[24] Metcalf, M., Reid, J.: Fortran 90 Explained. Oxford Science Publications, 1990

[25] MPL. MasPar Computer Corporation. MasPar Parallel Application Language (MPL) Reference Manual, September 1990

[26] Occam. Prentice Hall, Englewood Cliffs, New Jersey. INMOS Limited: Occam Programming Manual, 1984

[27] Philippsen, M., Tichy, W.F.: Compiling for massively parallel machines. In: Proc. of the Workshop on Code Generation, Schloss Dagstuhl, May 20-24 1991, Springer to appear

[28] Rosing, M., Schnabel, R., Weaver, R.: Dino: Summary and example. In: Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications, pp. 472–481. ACM Press, New York, February 26 – March 2 1988

[29] Sankoff, D., Kruskal, J.B. (eds): Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, 1983

[30] Smith, B.J.: Architecture and applications of the HEP multiprocessor computer system. In: Real Time Signal Processing IV, Proceedings of SPIE, pp. 241–248. International Society for Optical Engineering, 1981

[31] Tichy, W.F., Herter, C.G.: Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Interner Bericht Nr. 4/90, Universität Karlsruhe, Fakultät für Informatik, Januar 1990

[32] Tichy, W.F., Philippsen, M.: Hochgradiger Parallelismus. Interner Bericht Nr. 14/91, Universität Karlsruhe, Fakultät für Informatik, August 1991