Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 81 (1990)

Heft: 17

Artikel: Ein Parallel-Computer mit verteiltem Speicher : das K2-Projekt

Autor: Annaratone, Marco / Bonsen, Georg zur / Fillo, Marco

DOI: https://doi.org/10.5169/seals-903149

Ein Parallel-Computer mit verteiltem Speicher

Das K2-Projekt

Marco Annaratone, Georg zur Bonsen, Marco Fillo, Kiyoshi Nakabayashi, Claude Pommerell, Roland Rühl, Peter Steiner und Marc Viredaz

Vor zwei Jahren wurde an der ETH Zürich das K2-Projekt gestartet mit dem Ziel, einen parallelen Prozessor mit verteiltem Speicher zu entwickeln. Die K2-Architektur unterstützt einen automatisch parallelisierenden Fortran-Compiler und ein virtuelles Timesharing-Multiuser Multitasking-Betriebssystem. Der vorliegende Bericht beschreibt vier Hauptgesichtspunkte des Projekts: den Aufbau der Maschine, seinen parallelisierenden Compiler, das Betriebssystem sowie speziell für diese Architektur entwickelte Finite-Elemente-Algorithmen.

Le projet K2, commencé il y a deux ans, a pour but la conception et la réalisation d'un ordinateur parallèle à mémoire distribuée, dont l'architecture supporte, de manière efficace, un compilateur Fortran effectuant automatiquement la parallélisation, et un système d'exploitation multi-tâche, multi-utilisateur, en temps partagé. Ce papier présente les quatre aspects, les plus importants, de ce projet, c'est-à-dire, l'architecture de la machine, son compilateur parallélisant, son système d'exploitation et les algorithmes, utilisant la méthode des éléments finis, développés pour cette architecture.

Adresse der Autoren

Prof. Dr. *Marco Annaratone, Georg zur Bonsen, Marco Fillo, Claude Pommerell, Roland Rühl, Peter Steiner, Marc Viredaz,* Institut für Integrierte Systeme, ETH Zürich, 8092 Zürich, und *Kiyoshi Nakabayashi,* NTT Communications and Information Processing Laboratories, Tokyo 180, Japan

Die bisher entwickelten Prozessoren mit verteiltem Speicher (Distributed Memory Parallel Processors oder DMPPS) basieren auf verschiedenen Netzwerktopologien, wie z.B. Tori, Hypercubes und Linear Arrays. Auch unterscheiden sie sich im verfügbaren Grad der Parallelität einerseits und der Leistung eines Einzelprozessors andererseits. Zum Beispiel verfügt der MIMD iPSC/2 über 64 Vektorprozessoren und die SIMD Connection Machine CM-2 über 65,536 einfache Prozessoren. Effiziente Netzwerktopologien und Kommunikationsmechanismen wurden in den vergangenen Jahren untersucht. Im Vergleich dazu ist die Entwicklung von Systemsoftware im Rückstand. Wir halten dies für einen der Gründe, dass DMPPs noch keine kommerzielle Verbreitung finden. Wir konzentrieren uns deshalb auf die Entwicklung eines automatisch parallelisierenden Compilers (APC), eines virtuellen Timesharing-Betriebssystems und eines interaktiven symbolischen Debuggers. Die Berücksichtigung dieser Software beeinflusst den Hardware-Entwurf eines DMPP wesentlich und führt zu einer neuen Beurteilung der oben genannten Architekturparameter.

Aus Platzgründen können wir auf das Projekt nicht im Detail eingehen und werden K2 und seine Systemsoftware nur allgemein beschreiben. Die Architektur wird genauer in [1; 2] beschrieben. In [3] wird näher auf den automatisch parallelisierenden Compiler namens Oxygen eingegangen.

Übersicht über die K2-Architektur

Der Entwurf von K2 unterstützt die Ausführung von Anwendungsprogrammen und Systemsoftware. Bisherige Untersuchungen von Netzwerktopologien für parallele Anwendungssoftware sagen wenig über die Eignung dieser Topologien für parallele Systemsoftware aus. Wir haben uns deshalb für eine Topologie entschieden, deren Eignung für Anwendungssoftware wohlbekannt ist, und diese zur besseren Unterstützung von Systemaktivitäten modifiziert.

In Bild 1 ist der für Benutzerprogramme sichtbare Teil der K2-Architektur abgebildet. In einem Torus sind *Computation Nodes* (CN) bidirektional durch Paare von 32-Bit-Leitungen einschliesslich Fifos, den sogenannten *User Channels,* verbunden. Der Grundriss des K2 auf Systemebene ist in Bild 2 dargestellt. Jede Reihe und jede Spalte der CN ist mit je einem *Input-Output* Node (ION) verbunden, an

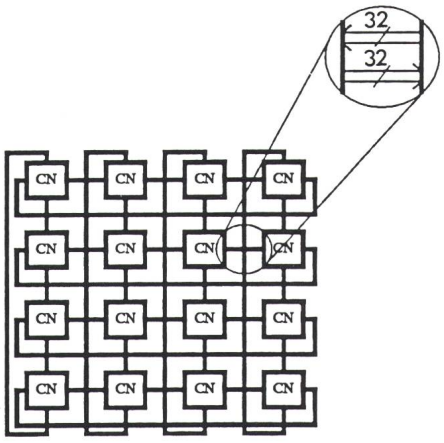

Bild 1 Benutzersicht des K2

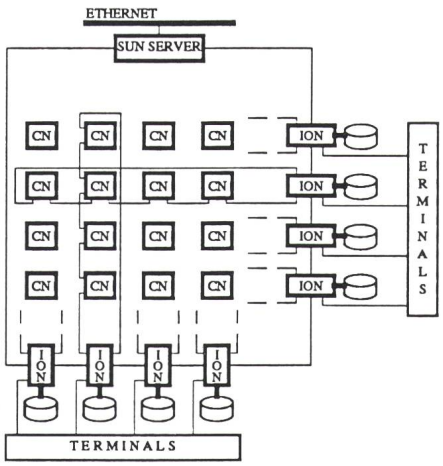

Bild 2 Systemsicht des K2

Parameter	User Channels	System Channels
Verwendung	Benutzerkommunikation	Systemkommunikation
Austausch von	Rohdaten	Meldungen
Routing	manuell (durch CPU)	automatisch (durch SNIK)
Medium	Parallele Leitungen	Koaxialkabel und Glasfaser
Typ	blockierend und nicht blockierend	nicht blockierend
max. Datenrate pro Kanal	$400Mb/s$	$100Mb/s$
Latenzzeiten:		
Register zu Register	$160ns$	
SNIK zu SNIK		$1.5\mu s$ (min)

Tabelle I Eigenschaften der User- und System Channels

Einheit	Anzahl Chips	Fläche (cm^2)	max. Leistungsverbrauch (W)
PE-CN	160	846	115
PE-ION	190	860	115
SNIK	85	754	30
User Channels	120	517	50
Disk Controller	48	308	35
CN	365	2117	195
ION	323	1922	180

Tabelle II Abmessungen und Leistungsverbrauch des K2-Prototyps

dem Terminals und Plattenlaufwerke angeschlossen sind. Diese Verbindung zwischen ION und Reihen bzw. Spalten von CNs, der sogenannte *System Channel,* ist als Token-Ring implementiert. Ein identischer Token-Ring verbindet alle IONs mit einem Ethernet-Gateway.

Alle Knoten (CN und ION) sind für die Ausführung von Benutzer- und Systemprozessen vorgesehen. Die CNs übernehmen die rechenintensiven parallelen Aufgaben, die IONs die Bearbeitung von interaktiven seriellen Programmen, wie z.B. Editoren und E-Mail. Obwohl die Aktivitäten des Betriebssystems auf alle Knoten verteilt werden, erfüllen die IONs hauptsächlich die Funktion von File-Servern und intelligenten *Disk Caches.* Die Eigenschaften der User- und System Channels sind in der Tabelle I zusammengefasst.

Die Hardware

Der K2 besteht aus zwei verschiedenen Bauelementen:

- Der *Computation Node (CN)* enthält ein Prozessor-Element (PE), je vier ausgehende und ankommende, 32 Bit breite User Channels und einen Serial Network Interface Controller oder SNIK, der die Kommunikation durch die System Channels steuert.
- Der *I/O-Node (ION)* enthält ein Prozessor-Element, einen Disk-Controller und Terminal-Schnittstellen, aber keine User Channels.

Die Prozessor-Elemente des ION und des CN unterscheiden sich nur bezüglich Speichergrösse. Der lokale Speicher des ION ist viermal grösser, da mit einem intelligenten Disk-Cache-Mechanismus die Leistungsfähigkeit des Seitenwechselverfahrens des virtuellen Betriebssystems erhöht werden soll. Insgesamt mussten demnach die vier Grundeinheiten PE, User Channel, Snik und Disk Controller entwikkelt werden. Beim hier vorgestellten Rechner handelt es sich um den K2-Prototyp, der einen vereinfachten Aufbau aufweist. Erstens wird nur ein 4-mal-4-Torus realisiert, und zweitens sind die PEs nicht mit zusätzlichen Cache-Speichern ausgerüstet.

Die Bilder 3a und 3b zeigen Blockdiagramme des CN und des ION. Der CN besteht aus einem Mikroprozessor AMD Am29000 mit einem Fliesskomma-Koprozessor AMD AM29027 (FPC), einem getrennten Instruktions- und Datenspeicher (0,5 MByte bzw. 2 MByte) mit Fehlererkennung und -korrektur, vier Paaren von 32 Bit breiten User Channels und einem Serial Network Interface Controller (SNIK). Die Architektur des ION gleicht bis auf zwei Ausnahmen jener des CN. Erstens fehlen die User Channels, und zweitens wurde die Kapazität der Instruktions- und Datenspeicher auf 2 MByte beziehungsweise 8 MByte erhöht. Ein intelligenter Mass Storage Controller oder MSC ist einerseits mit dem Prozessor-Element über einen Dual-Port-Speicher verbunden und verfügt andererseits über eine SCSI-Schnittstelle (Small Computer System Interface) mit einer Spitzen-Übertragungsrate von 4 Mbyte/s.

Die Abmessungen und der Energieverbrauch der verschiedenen Einheiten sind in Tabelle II zusammengefasst. Bild 4 zeigt eine bestückte CN-

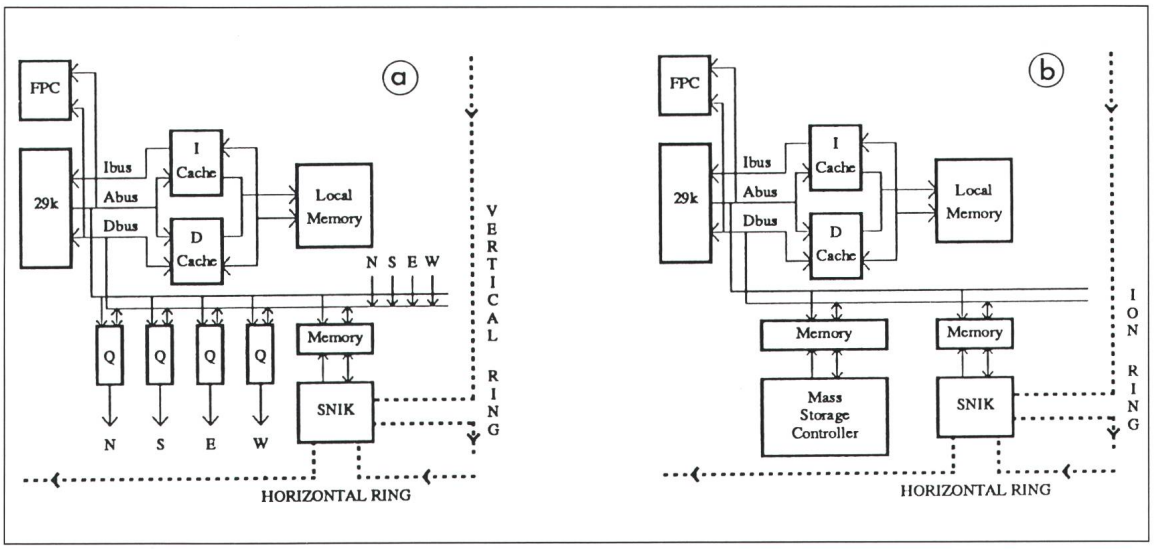

Bild 3
Blockdiagramm des CN (a) und ION (b)
Der K2-Prototyp enthält keine Cache-Speicher
CN Computation Node
ION Input-Output Node

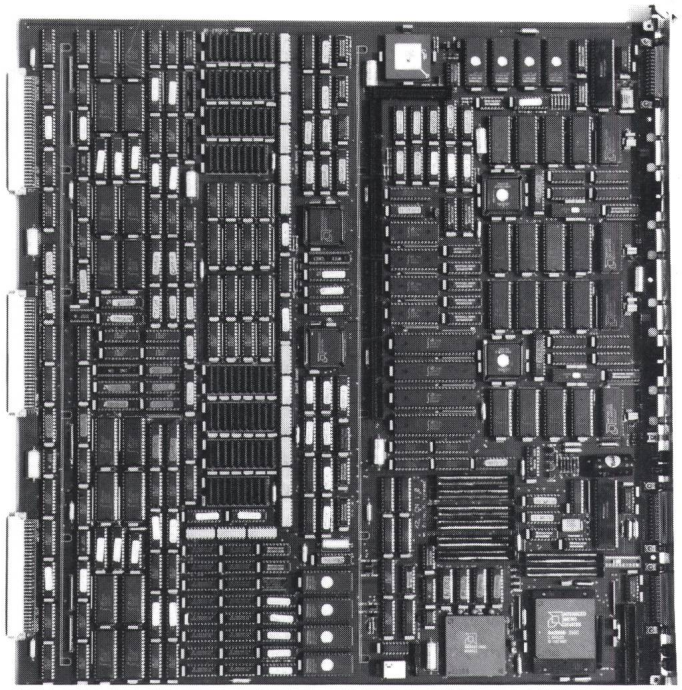

Bild 4
Die CN-Leiterplatte

Platine. Die Platinen wurden ausserhalb des Hauses hergestellt. Der Entwurf, die grafische Definition der Schaltungen und die Plazierung der Chips und die Leitungsführung wurden in unserem Labor durchgeführt.

Die Wahl des Prozessors

Die Wahl des Prozessors und der Struktur des lokalen Speichersystems waren die Hauptaufgaben beim Entwurf des Prozessor-Elements (PE). Wir haben uns entschieden, einen kommerziell erhältlichen und nicht einen kundenspezifischen Mikroprozessor einzusetzen, weil der erstere sowohl für den Hardware-Entwurf (es existieren Chips für Unterstützungsfunktionen und Entwicklungswerkzeuge) als auch für die Software-Entwicklung (Verfügbarkeit von Assembler, Compiler und Debugger) klare Vorteile aufweist.

Zum Zeitpunkt der Wahl des Mikroprozessors (Mai 1988) waren vier Hochleistungsprozessoren auf dem Markt oder in Markteinführung begriffen: der AMD Am29000, der Motorola MC88100, der Sparc und der Mips R2000. Die letzten beiden wurden aus nichttechnischen Gründen nicht weiter in Betracht gezogen. Die verbleibenden zwei Prozessoren, der Am29000 und der MC88100, verfolgen unterschiedliche Konzepte, deren detailliertere Untersuchung ausserhalb des Rahmens dieses Artikels liegt. Die Tabelle III zeigt die Prozessorarchitektur-Eigenschaften, welche die Evaluation beeinflusst haben.

Die Schnittstelle zu den User Channels hat schliesslich zugunsten des Am29000 den Ausschlag gegeben. Die Architektur des Prozessors sollte nämlich einen schnellen Zugriffsmechanismus auf die User Channels (wobei die Zugriffe vom Cache nicht zu behandeln sind) und ihr rasches Blockieren und Deblockieren ermöglichen.

Mit dem Am29000 verursacht die Abbildung der *User Channels* auf den virtuellen Adressraum keine Leistungseinbusse, weil der in den Prozessor integrierte Translation Lookaside Buffer eine schnelle Adressübersetzung in einer Pipelinestufe durchführt. Beim MC88100 hingegen verlangsamt die Abbildung der User Channels auf virtuelle Adressen den Zugriff, da die Adressierung durch eine auf einem separaten Chip untergebrachte Speicherverwaltungseinheit (CMMU)

Abkürzungen

APC	Automatically Parallelizing Compiler
CN	Computation Nodes
CMMU	Cache and Memory Management Unit
DMPP	Distributed Memory Parallel Processor
DRAM	Dynamic Random Access Memory
FIFO	First in First out
FPU	Floating Point Unit
ION	Input-Output Node
MIMDS	Multiple Instruction Multiple Data Stream
MIPS	Million Instructions per Second
MMU	Memory Management Unit
MSC	Mass Storage Controller
PE	Processor Element
SCSI	Small Computer System Interface
SIMDS	Single Instruction Multiple Data Stream
SNIK	Serial Network Interface Controller
SRAM	Static RAM
TAXI	Transparent Asynchronous Receiver/Transmitter Interface

PARAMETER	AM29000	MC88100
Taktfrequenz	25MHz	20MHz
Bus-Architektur	3 Busse	4 Busse
Bus-Protokoll	einfach, pipelined, Burst-Mode	synchron
On-chip Register	192	32
Instruktions-Cache	8 KByte, 2-Weg set-assoziativ[1]	16 ÷ 64 KByte, 4-Weg set-assoziativ Off-chip (88200)
Daten-Cache	8 KByte, 2-Weg set-assoziativ[1]	16 ÷ 64 KByte, 4-Weg set-assoziativ, off-chip (88200)
Branch-Target Buffer	128 Adressen, on-chip	nicht vorhanden
FPU	off-Chip (29027)	on-chip
MMU	on-Chip	off-chip (88200)

Tabelle III Vergleich der technischen Daten der Prozessoren AMD, Am29000 und Motorola MC88100

[1] Nach Einführung dieses Prozessors hat AMD die weitere Entwicklung separater Cache-Chips eingestellt

Speicher-Zugriff	Zugriffszyklen			
	I&D Cache	I Cache	VRAM	split-memory
Instr. b.i.	2	2	6	5
Instr. b.s.	1	1	1	2
Daten b.i.	2	6	4	6
Daten b.s.	1	1	1	2

Test programm	Leistung (Am29000 MIPS)			
	I&D Cache	I Cache	VRAM	split-memory
B1	23.02	20.58	17.38	13.48
B2	23.92	21.48	19.94	13.48
B3	23.23	19.12	18.35	14.10
B4	20.15	14.12	14.73	12.08
B5	22.87	20.71	18.43	13.81

Tabelle IV Leistungsvergleich der vier Speicheranordnungen

Instr. Burst Initialisation	Anzahl der Zyklen, um die erste Instruktion eines Instruktionsblockes zu laden.
Instr. Burst Steady-State	Anzahl der Zyklen, um eine Instruktion im Burst-Mode zu lesen.
Daten Burst Initialisation	Anzahl der Zyklen, um das erste Wort eines Datenblockes zu laden.
Daten Burst Steady-State	Anzahl der Zyklen, um auf ein Wort im Burst-Mode zuzugreifen

hindurch erfolgen muss. Ausserdem bewirkt das Bus-Protokoll zwischen Prozessor und CMMU eine Verlangsamung für nicht vom Cache zu behandelnde Zugriffe um mindestens 5 Zyklen.

Ein weiterer Gesichtspunkt ergab sich aus der Effizienz der Ausnahme-Behandlung bei blockiertem User Channel. Während der Am29000 einen Hardware-Mechanismus für die Fortsetzung der Ausführung einer abgebrochenen Instruktion aufweist, muss beim MC88100 auf Software-Emulation zurückgegriffen werden.

Der Entwurf des Speichersystems

Vier verschiedene Möglichkeiten standen beim Entwurf des Speichersystems zur Diskussion:

- Ein System, basierend auf getrenntem Instruktions- und Datencache mit lokalem Zweiweg-Interleaved-Dram
- Ein lokaler dynamischer Speicher mit Instruktions-Cache (nachfolgend I-Cache)
- Ein Video-Dram ohne Caches
- Getrennte Dram-Bänke für Instruktionen und Daten, keine Caches (nachfolgend Split Memory)

Die vier Speicherstrukturen wurden unter Benutzung eines Am29000-Instruktionssatz-Simulators verglichen. Die Tabelle IV stellt die angenommenen Speicherzugriffszeiten dar. Mit diesen Eingangsdaten wurden auf dem Simulator die in der gleichen Tabelle (Am29000 MIPS) gezeigten Geschwindigkeiten (in Mips) gemessen. Die fünf Benchmark-Programme (B1 bis B5) stammen ausschnittweise von auf dem K2-Simulator entwickelten parallelen Programmen [4]. Angaben für das I&D-Cache-System sind eher optimistisch, da der Simulator auf Caches basierende Systeme nicht vollständig modellieren kann. Die sich auf das I-Cache-System beziehenden Angaben sind realistischer. Tatsächlich konnten wir zeigen, dass Trefferquoten nahe 100% selbst bei kleinen Cachegrössen unabhängig von der Cache-Organisation erzielt werden [5]. Um die Anzahl Bausteine zu begrenzen und weil der Cache-Chip Am29062 noch nicht verfügbar ist, wurden die ersten beiden Entwurfsalternativen fallengelassen. Eine Lösung, basierend auf VRAMs, ist teuer und benötigt eine komplexe Fehlerkorrekturschaltung. Deswegen wurde die Entwurfsalternative *Split Memory* ausgewählt.

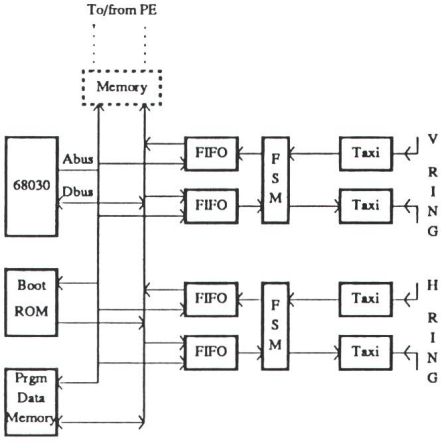

Bild 5 Blockdiagramm des Serial Network Interface Controllers

Serial Network Interface Controller

Bild 5 zeigt das Blockdiagramm des Serial Network Interface Controllers (SNIK). Der Snik besteht aus einem Motorola-Mikroprozessor MC68030, der mit eigenem Boot-ROM, Programm- und Datenspeicher ausgerüstet ist. Der Mikroprozessor bietet im Vergleich zu einem Controller mehr Spielraum beim Entwickeln und Testen von Kommunikationsprotokollen. Die physikalische Verbindung des MC68030 zu den System Channels wird mit zwei Paaren AMD- Taxi-Bausteinen (Transparent Asynchronous Transmitter/Receiver Interfaces) hergestellt. Diese integrierten Schaltungen wandeln von 8 Bit parallel zu seriell beim Senden, und von seriell zu parallel beim Empfangen. Die Sniks werden mit Koaxial- oder Glasfaserkabeln verbunden.

Der MC68030 leitet Pakete weiter, vermittelt sie zwischen horizontalem und vertikalem Ring (Corner Turn), berechnet und prüft Paritätssummen und sendet pro erhaltenes Paket eine Empfangsbestätigung. Fifos zwischen dem MC68030 und dem Taxi-Baustein puffern den Paketfluss. Ein komplexer Zustandsautomat zwischen Fifo und Taxi-Bausteinen interpretiert ankommende Pakete und puffert sie, falls sie für den lokalen Prozessor bestimmt sind (oder ein Corner Turn vollzogen werden muss). Der sendende Knoten sorgt für das Entfernen rücklaufender Pakete. Im Zustandsautomaten sind das Token-Ring-Protokoll und die Fehlerdetektion implementiert.

Das Bild 6 zeigt quantitativ einen einzelnen K2-Token-Ring. Bild 6a zeigt Kurven für den effektiven Durchsatz, während Bild (6b) die Latenzzeit, die im schlechtesten Fall auftritt, darstellt. Mit einer Paketgrösse von 512 Byte wird bereits ein Durchsatz nahe der Bandbreite des physikalischen Mediums (12,5 Mbyte/s) erreicht. Anderseits ist die Latenzzeit bei dieser Paketgrösse untragbar, da sie 0,5 ms oder länger dauern kann. Diese Angaben geben eine obere Schranke der Leistung des Token-Rings. Der Quotient aus effektivem Durchsatz

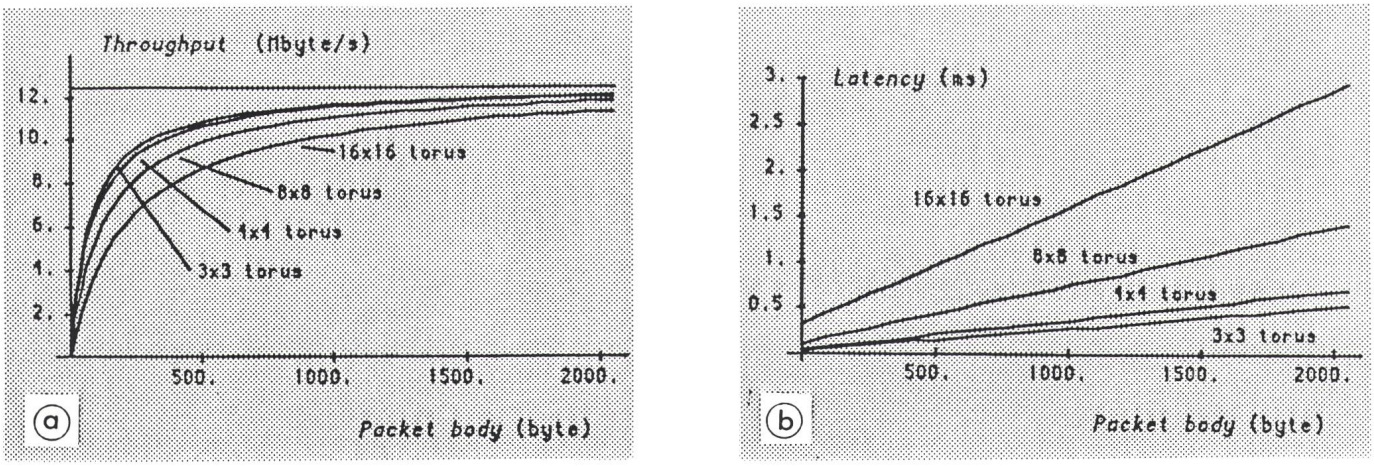

Bild 6 Datenrate auf einem einzelnen Token Ring

Die effektive Datenrate (a) und Latenzzeiten im ungünstigsten Fall (b) werden gegenüber Paketgrössen für 3×3, 4×4, 8×8 und 16×16 K2-Torus illustriert. Jeder Ring enthält 4, 5, 9 respektive 17 Prozessoren

und Latenzzeit erreicht bei einer Paketgrösse von etwa 92 Byte sein Maximum.

Der Oxygen Compiler

Die Forschung auf dem Gebiet der automatisch parallelisierenden Compiler (APC) konzentrierte sich bisher auf Multiprozessoren mit gemeinsamem Speicher. Die Prozessoren dieser parallelen Rechner sind meistens mit einem gemeinsamen Bus verbunden. Da der Bus eine begrenzte Bandbreite hat, wurden komplizierte Cache-Mechanismen entwickelt, um Datenlokalität auszunutzen. Das sich ergebende Kohärenzproblem wird zur Programmlaufzeit mit von der Hardware unterstützten Cache-Protokollen aufgelöst.

APCs (Automatically Parallelizing Compilers) für diese Maschinen restrukturieren und parallelisieren Code zur Compilierzeit. Der APC führt keine prozessorbezogene Datenallokation durch. Die Datenallokation durch die lokalen Caches ist damit für den Benutzer *und* für den Compiler transparent. Es wird üblicherweise ein Satz von Heuristiken benutzt, um das serielle Programm zur Compilierzeit zu parallelisieren. Beispielsweise können Datenabhängigkeiten innerhalb einer *DO*-Schleife zur Compilierzeit ausgewertet werden, wenn in zwei Ausdrücken innerhalb der Schleife ein Fortran-Array mit verschiedenen linearen Funktionen des Schleifenindex indiziert wird [6]. Natürlich können nicht alle Datenabhängigkeiten so aufgelöst werden.

Ein DMPP (Distributed Memory Parallel Processor) wie K2 unterscheidet sich von Multiprozessoren mit gemeinsamem Speicher im Hinblick auf einen zu entwickelnden automatisch parallelisierenden Compiler in folgenden beiden Punkten:

1. Ein APC für DMPPs muss Datenstrukturen auf die verfügbaren lokalen Speicher aufteilen (Domain Decomposition).
2. Der Aufwand, eine von mehreren Prozessoren benutzte Variable anzusprechen, ist abhängig von der Netzwerkposition des Prozessors, auf dem die Variable alloziert wurde.

Da DMPPs keine Hardware aufweisen, um Speicherkonflikte zu lösen, analysiert Oxygen Datenabhängigkeiten zum Teil zur Programmlaufzeit. Die folgenden Abschnitte zeigen, wie dadurch die Geschwindigkeit des ausführbaren Codes beeinflusst wird.

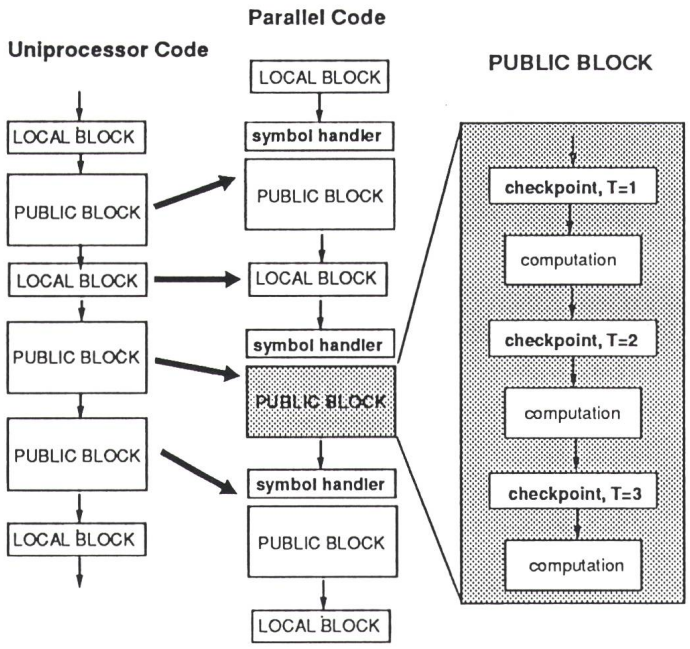

Bild 7
Die Aufteilung eines seriellen Programms mit Oxygen

Oxygen-Grundkonzept

Der Entwurf von Oxygen verläuft in zwei Phasen. Erst wird ein Compiler mit Direktiven implementiert, dann wird ein Präprozessor entwickelt, der diese Direktiven automatisch generiert.

Oxygen setzt voraus, dass jedes Programm in Code-Blöcke zweier verschiedener Klassen zerlegt werden kann:

- die Klasse der *lokalen Blöcke* impliziert keine Interprozessorkommunikation.
- Die Benutzung bestimmter Variablentypen in *öffentlichen Blöcken* impliziert die Generierung von Kommunikationsprimitiven zur Laufzeit.

Das Bild 7 zeigt, wie ein serielles Programm mit Oxygen zerlegt wird. Die erste Spalte der Figur zeigt die Zer-

legung des Pprogramms mit Compilerdirektiven in lokale und öffentliche Blöcke. Die zweite Spalte zeigt den C++-Code, der für alle Prozessoren gemeinsam von Oxygen erzeugt wird. Jedem öffentlichen Block wird ein zusätzlicher Code - der sogenannte *Symbol-Handler* - vorangestellt. Dieser Symbol-Handler generiert zur Laufzeit Datenstrukturen, die der übersetzte Block zur Generierung und Ausführung von Kommunikationsprimitiven (SEND/RECEIVE) verwendet. Die Kommunikationsprimitiven werden nur zu bestimmten *Communication Checkpoints* ausgeführt. Die Aufteilung jedes öffentlichen Blocks durch diese Checkpoints wird in der dritten Spalte der Graphik dargestellt. Dieses Modell zeigt, dass zur Laufzeit Datenabhängigkeiten im Symbol-Handler analysiert werden.

Beispielprogramme und Benchmarks

Wir zeigen hier Ergebnisse von sechs Testprogrammen: die allgemeine FFT (AFFT), Gauss-Elimination mit partiellem Pivoting, Orthes (Transformation einer Matrix in Hessenbergform), die Lanczos-Iteration für unsymmetrische Matrizen, ein Ausschnitt aus der Eispack-SVD()-Routine sowie die Lösung einer zweidimensionalen Differentialgleichung, wie sie bei der Berechnung der Dynamik stationärer laminarer Strömungen vorkommt.

Allgemeine FFT

In ihrer ersten Formulierung des FFT-Algorithmus beschreiben Cooley und Tukey die FFT für jede Problemgrösse *N*. Wir haben diesen Algorithmus auf Oxygen implementiert und erhalten Speedups, wie in Tabelle V gezeigt. Die Problemgrösse bezieht sich auf die Länge des komplexen Eingangsvektors der AFFT. In diesem Programm werden Fortran-Arrays mit komplexen Integer-Ausdrücken indiziert. Die Auswertung dieser Ausdrükke zur Compilierzeit durch konventionelle Datenabhängigkeitsanalyse ist unmöglich, da der Wert der Ausdrücke von nur zur Laufzeit vorhandener Information (nämlich der Primfaktorzerlegung von *N*) abhängt. Dies gilt nicht für die einfache FFT, bei welcher die Länge des Datenvektors ein Vielfaches von 2 ist. Deswegen können Kommunikationsprimitive *nur* zur Laufzeit generiert werden.

Tabelle V
Zusammenfassung der Speedups für Oxygen-Implementationen der allgemeinen FFT und der Lösung der Differentialgleichung aus der Strömungsdynamik

Algorithmus	*Problemgrösse*	*Torusgrösse*	*Speedup*
AFFT	10000	16	12.31
AFFT	7776	36	13.10
AFFT	10000	64	12.47
Fluid	200 × 100	16	11.0
Fluid	200 × 100	36	15.1
Fluid	200 × 100	64	12.8

Lineare Algebra

Die Testprogramme für Gauss-Elimination, Orthes, Lanczos-Iteration und SVD()-Routine beziehen sich auf die Implementation von Standardalgorithmen der numerischen linearen Algebra (diese Algorithmen werden zum Beispiel von Wilkinson und Reinsch [7] beschrieben). Speedup-Resultate mit Oxygen zeigt das Bild 8.

Strömungsdynamik

Die Auswertung des Verhaltens einer Tragfläche bei Unterschall-, Schall- und Überschallgeschwindigkeit wird nach der Methode von Murman und Cole [8] durchgeführt. Dabei wird eine zweidimensionale stationäre laminare Strömungsgleichung gelöst. Näheres zu dem Programm kann in [9] gefunden werden. Speedup-Ergebnisse sind in Tabelle V zu finden. Die Problemgrösse bezieht sich auf die Grösse des Diskretisierungsgitters des auf SOR (Successive Over-Relaxation) basierenden Algorithmus.

Das Betriebssystem des K2

Unter dem Namen *Chagori* wird ein virtuelles Mehrbenutzer-Betriebssystem für den K2 entwickelt. Ein solches ist auf Uniprozessoren und auf Parallelrechnern mit gemeinsamem Speicher inzwischen zur Selbstverständlichkeit geworden. Für DMPPs wird ein solches Betriebssystem ein grösseres Anwendungsgebiet erschliessen und ist damit eine Aufgabe, die ebenfalls weiterer Forschung bedarf.

Die Benützung des K2 soll zukünftig durch die Überwindung der Auftrennung in einen Wirtsrechner und einen daran angeschlossenen Parallelrechner vereinfacht werden. Der Benutzer beschränkt sich auf die vom Betriebssystem angebotene Abstraktion des Prozesses und, spezifisch für den K2, der parallelen Prozessgruppe. Einzelprozesse sowie Prozesse einer Gruppe stellen die auf allen Prozessoren des K2 gleichartige Umgebung zur Ausführung von Programmen dar. Diese Umgebung umfasst einen sehr grossen

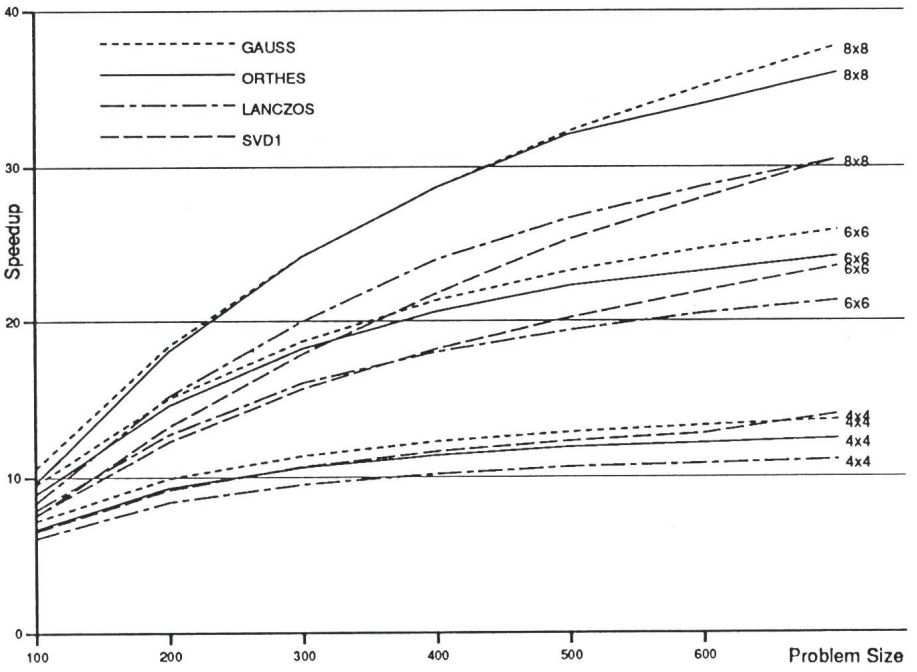

Bild 8 Speedup-Ergebnisse für die Gauss-Elimination, Orthes, SVD1 und die Lanczos-Iteration auf einem Torus mit 4×4, 6×6 und 8×8 Prozessoren

virtuellen Adressraum und die prozessorunabhängige Verfügbarkeit aller Systemdienste, einschliesslich der interaktiven Ein- und Ausgabe sowie des Zugriffes auf das verteilte Dateisystem.

Parallele Prozessgruppen belegen den Torus nach einem Time-Sharing-Schema. Charakteristisch für Prozessgruppen ist, dass topologisch benachbarte Prozesse direkt über die User Channels miteinander kommunizieren und dass auf verteilte Dateien parallel zugegriffen werden kann. Mit der Prozessgruppe erzeugt das Betriebssystem gegenüber dem Benutzer die Illusion, jederzeit über einen unabhängigen Parallelrechner zu verfügen. Mit diesem Konzept können bei vertretbaren Kosten parallele Programme entwikkelt und unter einem Debugger interaktiv getestet werden.

In mancher Hinsicht gleicht Chagori verteilten Betriebssystemen wie Amoeba [10] oder Mach [11], an die es auch angelehnt ist. Die einzelnen Prozessoren laufen unter der Kontrolle eines einfachen Betriebssystemkerns, dessen Funktion sich darauf beschränkt, Prozesse und eine netzwerktaugliche Methode der Interprozess-Kommunikation zu unterstützen. Alle übrigen Systemfunktionen können von darauf spezialisierten, auf beliebigen Prozessoren des Netzwerks lokalisierten Server-Prozessen transparent über das Netzwerk erbracht werden.

Im Gegensatz zu den für unregelmässige und grossräumige Netzwerke von unterschiedlichen Rechnern ausgelegten Betriebssystemen muss Chagori die besondere Architektur des K2 ausnützen können. So müssen z.B. alle Prozesse einer Gruppe simultan aktiviert und inaktiviert werden, um einerseits Kommunikation über die User Channels zu ermöglichen und anderseits Prozessgruppen gegeneinander vollständig zu isolieren.

Weiter soll das Dateisystem die Aufteilung von Dateien auf mehrere Platten und damit parallele Zugriffe auf verstreute Dateielemente erlauben. Dies ist einfach und effizient realisierbar bei Dateien, die aus Blöcken fester Länge bestehen, wie sie vom Betriebssystem selbst als Seitenwechselbereiche für die virtuellen Adressräume der Prozesse verwendet werden.

Dateien von Blöcken vorgegebener fester Länge stellen jedoch nicht ein allgemein geeignetes Modell für parallel zugreifbare Dateien dar. Verfahren wie die direkte Abbildung von Dateien in den virtuellen Adressraum der Prozesse durch das Betriebssystem führen auf DMPPs leicht zu nicht effizient lösbaren Kohärenzproblemen und damit zu einem Verlust der Vorteile dieser Rechnerarchitektur.

Diese Nachteile weist ein etwas allgemeineres Verfahren nicht auf, bei dem sogenannte *Agenten* Dateien in die virtuellen Adressräume einer parallelen Prozessgruppe abbilden. Dabei schliesst diese Abbildung auch eine Verteilung der Elemente jeder Datei auf die beteiligten Prozesse ein, wie sie die meisten parallelen Programme erfordern. Eine allfällige Datenkonversion kann damit auch durchgeführt werden. Vom Betriebssystem aus gesehen, sind Agenten von der Anwendung bezeichnete Prozesse, welche die Seitenfehlerbehandlung für die in Dateien abzubildenden Datensegmente vornehmen.

Lösen grosser linearer Gleichungssysteme mit schwach besetzten Matrizen

Die effiziente Lösung von grossen linearen Gleichungssystemen mit schwach besetzten Matrizen ist eine wesentliche Aufgabe bei vielen wissenschaftlichen Problemen und nimmt dabei normalerweise den grössten Teil der benötigten Rechenleistung und des Speicherbedarfs in Anspruch. Die Gleichungssysteme entstehen bei der Diskretisierung für die numerische Lösung von partiellen Differentialgleichungen (zum Beispiel mit Finiten Elementen) und weisen eine sehr unregelmässige Struktur der Nichtnull-Einträge auf.

Für Systeme mit einer sehr grossen Zahl von Unbekannten, wie sie in dreidimensionalen Problemen entstehen, sind die klassischen, auf Gausscher Elimination basierenden direkten Lösungsmethoden wegen ihres grossen Speicherbedarfs nicht mehr anwendbar. Statt dessen werden iterative Methoden verwendet, welche sich mit aufeinanderfolgenden Approximationen dem korrekten Lösungsvektor nähern. Am erfolgreichsten sind dabei die sogenannten *präkonditionierten konjugierten Gradientenmethoden*.

Parallelisierung solcher iterativer Methoden für unregelmässig schwach besetzte Systeme ist wesentlich schwieriger als für Matrizen mit regelmässiger Struktur. Insbesondere für Rechner mit verteiltem Speicher (DMPP) findet man bisher noch überhaupt keine Ansätze dazu in der Literatur. In unserer Arbeit hat sich gezeigt, dass bei der Implementierung für DMPPs neuartige Vorgehensweisen erforderlich sind, welche bei den bisherigen Realiserungen auf Vektorrechnern und Multiprozessoren mit gemeinsamem Speicher nicht notwendig waren.

Die oben erwähnten iterativen Methoden bestehen aus der wiederholten Anwendung einiger weniger Rechenschritte, welche sich in die vier folgenden Typen von Operationen einteilen lassen:

1. Lineare Operationen (Addition und Skalierung) auf Vektoren.
2. Skalare Multiplikation von Vektoren.
3. Multiplikation von schwach besetzten Matrizen mit Vektoren.
4. Lösung von linearen Gleichungssystemen mit schwach besetzten Dreiecksmatrizen.

Bei der effizienten Implementierung dieser Operationen auf DMPPs sind die folgenden Punkte wichtig:

Verteilung der Daten

Zum Erreichen einer gleichmässigen Verteilung der Rechenlast auf die einzelnen Prozessoren müssen die benötigten *Daten*, also die Unbekannten und die Nichtnull-Einträge der Matrizen, möglichst gleichmässig verteilt werden. Eine optimale Effizienz bei der Ausführung von linearen Operationen auf Vektoren lässt sich dadurch erreichen, dass alle Vektoren gleichmässig und auf die gleiche Art und Weise verteilt werden. Die für die Berechnung eines Skalarproduktes von Vektoren benötigte globale Synchronisation, welche bei einigen Parallelrechnern einen Flaschenhals darstellt, erwies sich aufgrund der schnellen Kommunikationskanäle auf K2 als unproblematisch [2].

Abbildung des Problemgraphen auf die Rechnertopologie: Mapping

Eine schwachbesetzte Matrix wird üblicherweise mit einem *Graphen* gleichgesetzt. Ein Nichtnull-Eintrag a_{ij} ausserhalb der Diagonale der Matrix entspricht einer Kante zwischen dem Knoten i und dem Knoten j des Graphen. Bei der Anwendung von schwach besetzten Matrizen bei der Lösung von partiellen Differentialgleichungen stellt dieser Graph das Diskretisierungsgitter dar. Bei einer Diskretisierung mit Finiten Elementen entsprechen die Ecken und Kanten der Elemente den Knoten und Kanten des Graphen.

Bei der oben erwähnten Verteilung der Vektoren werden die Knoten dieses Graphen auf die Prozessoren verteilt. Bei denjenigen Kanten des Graphen, welche Knoten verbinden, die nicht dem gleichen Prozessor zugeordnet sind, muss bei der Berechnung eines Matrix-Vektor-Produktes ein Datenaustausch stattfinden. Das *Mapping* hat zum Ziel, den Graphen so auf die Topologie des Rechners abzubilden, dass der Aufwand an Kommunikation für diesen Datenaustausch möglichst klein ist. Da die Aufgabe, eine optimale Lösung für diese Abbildung zu finden, NP-vollständig ist, bedarf es Heuristiken zum Finden einer möglichst guten Lösung.

Parallele Ausführung der Präkonditionierung: Coloring

Präkonditionierung ist eine wesentliche Voraussetzung für den effizienten Einsatz von iterativen Methoden, da sie die Konvergenz sehr stark beschleunigt und oftmals sogar erst ermöglicht. Für die besten der heute bekannten Präkonditionierer müssen in jeder Iteration zwei lineare Gleichungssysteme mit schwach besetzten Dreiecksmatrizen gelöst werden. Da diese Operation im allgemeinen als schlecht parallelisierbar gilt, wird bei vielen Supercomputer-Implementierungen auf schlechtere Präkonditionierer zurückgegriffen.

In unserer Arbeit konnten wir allerdings *Kolorierungsmethoden* entwikkeln, mit Hilfe derer bei den besten Präkonditionierern eine ähnlich gute Effizienz wie bei den oben erwähnten Matrix-Vektor-Multiplikationen erreicht werden konnte. Das Prinzip der Kolorierung ist, alle Knoten des Graphen zu bestimmen, deren Teil der Lösung im Dreieckssystem ohne weitere Kommunikation berechnet werden kann. Die Knoten des Graphen werden dazu in eine Reihe von Untermengen (Farben) eingeteilt. Datenaustausch findet nur zwischen Knoten verschiedener Farbe auf verschiedenen Prozessoren statt. Dabei ist hier eine gleichmässige Verteilung der Rechenlast besonders kritisch.

Ergebnisse

Während die Implementierung von iterativen Methoden für unregelmässige Probleme auf einem Rechner mit verteiltem Speicher an sich schon sehr anspruchsvoll ist, liegt der wesentliche wissenschaftliche Wert dieser Arbeit in der Entwicklung von neuen Parallelisierungsstrategien. Wir haben eine Reihe von neuen Methoden für das Mapping und das Coloring entwickelt und in der konkreten Anwendung bei der Simulation von Halbleiterstrukturen mit echten grossen Gleichungssystemen (mit bis zu 75 000 Unbekannten) evaluiert. Bei allen untersuchten Problemen hat sich dabei gezeigt, dass die iterativen Methoden auf den 64 Prozessoren des K2-Torus zwischen 42- und 54mal schneller als auf Einzelprozessoren ausgeführt werden konnten, was einer für unregelmässige Probleme ungewöhnlich hohen Ausbeute von 65–85% entspricht. Die wenigen in der Literatur erwähnten, aber nie auf realistischen Problemen erprobten Parallelisierungsstrategien waren ausnahmslos unseren neuentwickelten Strategien unterlegen. Eine detailliertere Beschreibung der Problemstellung, der Lösungstrategien und ihrer Auswertung wurde kürzlich publiziert [12].

Projektstatus

Die Hardware des Rechners ist vollständig entworfen, und die einzelnen Einheiten wurden bei voller Geschwindigkeit getestet. Verschiedene CN- und ION-Platinen werden in den nächsten Monaten hergestellt und getestet. Die System-Backplane, das Chassis, Netzgeräte usw. werden gegenwärtig integriert. Alle Platinen sind so gross, dass sie Versteifungsstreben benötigen. Die Verteilung von 500 A bei 5 V innerhalb des Systems erfordert Stromschienen.

Dank

Das bisher Erreichte wäre nicht möglich gewesen ohne «a little help from our friends». Wir danken Constantine Polychronopoulos, CSRD, University of Illinois at Urbana-Champaign, und Thomas Gross, School of Computer Science, Carnegie Mellon University.

Am K2-Projekt mitgeholfen hat die Gruppe der folgenden Studenten: Felix Äbersold, Marc Brandis, Mirko Bulinsky, Pascal Dornier, Olivier Gemoets, Michael Halbherr, Alain Kaegi, Roland Lüthi, Alexandros Pappas, John Prior, Stefan Sieber, Markus Tresch und Othmar Truninger. Last but not least danken wir Norbert Felber von der VLSI-Gruppe.

Literatur

[1] *M. Annaratonne* a. o.: The K2 parallel processor: Architecture and hardware implementation. Proceedings of the 17th Symposium on Computer Architecture, Seattle, 28...31 May 1990.

[2] *M. Annaratone, C. Pommerell* and *R. Rühl:* Interprocessor communication speed and performance in distributed-memory parallel processors. Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem/Israel, May 28...June 1, 1989. ACM Sigarch Computer Architecture News 17 (1989) 3, p. 315...324.

[3] *M. Rühl* and *M. Annaratone:* Parallelization of Fortran code on distributed-memory parallel processors. Proceedings of the ACM Conference on Supercomputing, Amsterdam, June 1990.

[4] *P. Beadle, C. Pommerell* and *M. Annaratone:* K9: A simulator of distributed-memory parallel processors. Proceedings of the Conference on Supercomputing, Reno/Nevada, 13. November 1990.

[5] *M. Annaratone* and *R. Rühl:* Efficient cache organizations for distributed-memory parallel processors. Technical report. Zürich, Swiss Federal Institute of Technology, Integrated Systems Laboratory, 1989.

[6] *U. Banerjee:* Dependence analysis for supercomputing. Boston/Dordrecht/London, Kluwer Academic Publishers, 1988.

[7] *G.H. Golub* and *C. Reinisch:* Singular value decomposition and least-square solutions. In: Handbook for Automatic Computation. vol 2: J.H. Wilkinson and C. Reinsch: Linear Algebra, Berlin a.o., Springer-Verlag, 1971; p. 134...151.

[8] *E.M. Murman* and *J.D. Cole:* Calculation of plane steady transonic flows. Document D1-82-0943. Seattle, Boeing Scientific Research Laboratories/Flight Sciences Laboratory, January 1970.

[9] *J. Moran:* An introduction to theoretical and computational aerodynamics. New York a.o., John Wiley, 1984.

[10] *S.J. Mullender:* The Amoeba distributed operating system: Selected papers 1984...1987. CWI Tract 41. Amsterdam, Centrum voor Wiskunde en Informatica, 1987.

[11] *M. Accetta* a.o.: Mach: A new kernel foundation for unix development. Technical report. Pittsburg, Carnegie Mellon University, 1986.

[12] *C. Pommerell, M. Annaratone* and *W. Fichtner:* A set of new mapping and coloring heuritics for distributed-memory parallel processors. Copper Mountain Conference on Iterative Methods. Philadelphia, Society for Industrial and Applied Mathematics (SIAM), April 1990.