FACHBEREICH INFORMATIK

Universität Frankfurt

PARALLELISIERUNGSKONZEPT FÜR ATTEMPTO-2

J. LUTZ, R. BRAUSE, M. DAL CIN, TH. PHILIPP

INTERNER BERICHT 1/89

Parallelisierungskonzept für ATTEMPTO-2

J. Lutz, R. Brause, M. Dal Cin, Th. Philipp

J.W. Goethe-Universität Frankfurt
Praktische Informatik
Verteilte Systeme und Fehlertoleranz VSFT

Abstrakt

Diese Arbeit beschreibt ein Konzept zur Parallelisierung und Verteilung von Programmen in Attempto-2, einem homogenen, nachrichtengekoppelten Mehrrechnersystem, das Untersuchungen zu parallelen Systemen und Fehlertoleranz dient. Dieser Bericht erläutert die allgemeinen Voraussetzungen und beschreibt die Besonderheiten bei der Parallelprogrammierung im Kontext des Gesamtprojekts. Er berücksichtigt getroffene Einschränkungen und enthält Implementationshinweise.

Inhaltsverzeichnis

1. Einleitung
    1.1 Themenstellung
    1.2 Begriffe

2. Parallelisierungskonzepte
    2.1 Beschreibung paralleler Pfade
    2.2 Koordination paralleler Pfade
        2.2.1 Globale gemeinsame Variablen
        2.2.2 Botschaften

3. Parallelität für Attempto-2
    3.1 Forderungen und Ziele
    3.2 Auswahl paralleler Sprachkonstrukte
    3.3 Randbedingungen und Designentscheidungen

4. Programmgenerierung
    4.1 Sequentielle Programme
    4.2 Parallele Programme

5. Taskanstoß
    5.1 Prozeduraufruf in sequentiellen Programmen
    5.2 Remote Procedure Call
    5.3 Remote Service in Attempto-2

6. Schlußbetrachtung

Literaturverzeichnis

1. Einleitung

1.1 Themenstellung

Im Rahmen des Forschungsprojekts Attempto-2 beschäftigen wir uns mit paralleler Programmierung und Fehlertoleranz. Attempto-2 ist ein homogenes Mehrrechnersystem auf der Basis von Motorola MC68010 Single Board Computern (SBCs) mit gemeinsamem VME-Bus. Ein Teil des lokalen Speichers der einzelnen Rechner kann global über den Bus adressiert werden und dient der lose gekoppelten Kommunikation. Bedient wird der Attempto-Cluster über einen UNIX-Host. Die Attempto-2 Software soll Parallelprogrammierung, Test, Diagnose und Rekonfiguration auf diesem Mehrprozessorsystem ermöglichen. Dazu ist eine geeignete Programmierumgebung zu schaffen. Für Diagnose und Tests wird ein wissensbasiertes Expertensystem aufgebaut. Die für die parallele Programmierung notwendigen Rechner, sollen nämlich zusätzlich bei geringerer Parallelitätsanforderung der Verbesserung der Fehlertoleranz dienen. Zur Entwicklung des Gesamtsystems muß deshalb für Attempto-2 eine programmiersprachliche Beschreibung von Parallelität verfügbar sein, die schnell und einfach realisiert werden kann. Es sollen keine neuen parallelen Algorithmen für bereits gelöste Probleme gefunden werden müssen, sondern bestehende Programme leicht integriert werden können.

In diesem Beitrag soll nun die computerunterstützte Parallelisierung und Konfiguration von Programmen betrachtet werden. Zuerst werden einige Gesichtspunkte zur Syntax und Pragmatik von parallelen Programmen ausgeführt. Danach werden grundsätzliche Methoden und Überlegungen zur Parallelisierung von Programmen erläutert. Es folgen einfache Sprachmittel zur Beschreibung von Parallelität in *klassischen* Programmiersprachen. Dann werden die für Attempto-2 notwendigen und geeigneten Konstrukte ausgewählt. Abschließend werden die dadurch entstehenden Auswirkungen und Anforderungen an die Übersetzung, das Binden, die Laufzeitbibliothek und das Laden von parallelen Programmen beschrieben.

Dieser Bericht enstand nach und durch Diskussionen im Oberseminar:
Aktuelle Probleme der Parallelität und Fehlertoleranz im WS 87/88.

1.2 Begriffe

Um Mißverständnisse durch im folgenden verwendete Begriffe, die in der Literatur zum Teil andere Sinnbelegung haben, zu vermeiden, seien sie kurz in unserem Sinne eingeführt.

Programme sind in einer maschinell übersetzbaren *Programmiersprache* als Text niedergeschriebene Algorithmen zur Problemlösung auf einem Rechensystem. Teilaufgaben werden in Unterprogrammen gelöst, die als *Prozeduren* niedergeschrieben werden. Können solche Prozeduren unabhängig von anderen Programmteilen zeitlich parallel (*nebenläufig*) als Einheit bearbeitet werden, sollen sie als eine *Task* bezeichnet

werden. Der Aufruf einer solchen Prozedur, werde *Anstoß* zur Ausführung einer Task genannt. Die Ausführung selbst sei eine *Aktion*. Der Kontrollfluß des Programms kann sich in mehrere *parallele Pfade* aufteilen. Die sequentiellen Anweisungen eines parallelen Pfads werden immer zu Prozeduren, den Tasks, zusammengefaßt. Ein Abschnitt eines Programms, der mehrere parallele Pfade hat, wird als *paralleler Bereich* bezeichnet. Die Zahl der parallelen Pfade wird dabei die *Breite des parallelen Bereichs* genannt. Ein *Prozess* ist die vollständige Beschreibung einer Task aus der Sicht des Betriebssystems; ihm kann ein Prozessor zugeteilt werden. Ein *verteiltes Programm* besteht aus mehreren Prozessen und kann auf mehreren Prozessoren ablaufen.

Soll ein Programm nach der Übersetzung auf mehreren Prozessoren parallel (*nebenläufig*) ausgeführt werden können, muß die Parallelität entweder *implizit* gegeben sein oder *explizit* beschrieben werden.

Implizit heißt, daß die Übersetzungswerkzeuge voneinander unabhängige und dadurch möglicherweise nebenläufige Teilaufgaben erkennen und markieren müssen. Nachfolgende Werkzeuge müssen aus den markierten Teilstücken unabhängig ablauffähige Programmteilstücke (tasks) generieren, die dann auf mehreren Prozessoren als Prozesse laufen können.

Explizit heißt, daß die parallelen Pfade vom Programmierer durch geeignete Sprachmittel spezifiziert werden, so daß die nachfolgenden Werkzeuge daraus ein verteiltes Programm machen können. Meist wird dafür eine neue Programmiersprache entwickelt (Concurrent Pascal [Bri75] , OCCAM [May83]).

2. Parallelisierungskonzepte

Parallelität kann nach dem Umfang der nebenläufig ausführbaren Einzelaufgabe klassifiziert werden. Instruktionen können die kleinsten, parallel arbeitenden Einheiten (*grain modules*) bilden. Man spricht dann von feinkörniger (fine grained) Parallelität. Die Tasks (Prozeduren) bilden meist die nächste Stufe (medium grained). Als größte Einheit werden Programme (*jobs*) betrachtet (large grained). Im folgenden wollen wir immer Einheiten der mittleren Körnigkeit voraussetzen.

Zuerst sollen bekannte Sprachmittel zur Beschreibung paralleler Programme vorgestellt werden. Danach wenden wir uns den Konzepten zur Koordination paralleler Prozesse zu. Anschliessend erörtern wir Implementierungen der für uns in Frage kommenden Konzepte. Dabei betrachten wir insbesondere die Methoden der Koordination durch Synchronisation und Kommunikation verteilter Programme.

2.1 Beschreibung paralleler Pfade

Parallele Programme können durch unterschiedlichste Sprachmittel wie fork/join, cobegin/coend oder Coroutinen beschrieben werden.

Fork/join

Die Anweisungen *fork, join* [Den66] (s.a. UNIX fork(2),wait(2) [Uni79]) waren eine der ersten Sprachmittel zur Beschreibung von Parallelität. *Fork* spaltet den Befehlsfluß in parallele Pfade (tasks) auf. Mit *join* wird am Ende eines parallelen Bereichs gewartet. Da diese Sprachkonstrukte keine strukturierten Anweisungen sind und wie GOTO an fast allen Stellen im Programm syntaktisch zulässig sind, haben sie auch die entsprechenden Nachteile. Ein Beispiel dazu (P1,P2 sind tasks):

```
    ....
for i := 1 to 2
  fork(P1);
  fork(P2);         ---> P1;P2; P1;P2
end (* for *);
join(P2);                  (* warten auf P2 *)
    ....
(* andere 'joins' *)
```

Abb. 2.1: fork/join

In der zweimal durchlaufenen Schleife werden jeweils die Tasks P1,P2 erzeugt. Nach der Schleife wird auf die Terminierung von P2 gewartet. Auf welche der beiden mit P2 benannten Tasks wird gewartet ? Die Anwort auf diese Frage ist aus dem Programm nicht ableitbar. Diese fehlerhafte Konstruktion ist nur für stark analysierende Compiler erkennbar.

Cobegin/Coend

Cobegin,Coend [Dij68] sind die Schlüsselworte, welche einen *parallelen Bereich*, in dem die Tasks angestoßen werden, strukturiert beschreiben (s. Abb. 2.2). Anfang und Ende des parallelen Bereichs sind dadurch eindeutig festgelegt. Diese sprachliche Konstruktion einer zusammengesetzten Anweisung erlaubt es den Übersetzungswerkzeugen, lokale und gemeinsam benutzte (*shared*) globale Variablen zu unterscheiden. Da Synchronisation und Datenaustausch dem Anwender verborgen bleiben, muß die Konsistenz der gemeinsam genutzten (*shared*) Daten auf besondere Art und Weise vom System gesichert werden. Lösungsansätze dafür sind kritische Abschnitte, Semaphoren und Monitore.

Der Vorteil dieses Konstrukts besteht in der einfachen blockorientierten Darstellung von Parallelbereichen mit einem Eingang (Cobegin) und einem Ausgang (Coend). Ein Nachteil für Mehrprozessorsysteme ist der Aufwand für die versteckte Synchronisation und Kommunikation am Anfang und Ende des parallelen Bereichs. Gemeinsam genutzte globale Variablen müssen nach wie vor explizit geschützt werden. Eine fehlerhafte Benutzung kann wiederum nur von stark analysierenden Compilern erkannt werden.

```
cobegin
  P1;
  P2;              ------>  P1;P2;P3
  P3;
coend;
```

Abb. 2.2: cobegin/coend

Die Tasks P1,P2,P3 werden parallel ausgeführt. Nach dem Ende dieser Tasks wird nach dem *coend* fortgefahren.

Coroutinen

Die Aufgaben (tasks) werden als parameterlose Prozeduren definiert und dann als Coroutinen installiert. Die Coroutinen erhalten den Prozessor durch explizite Kontrollübergabe (Transfers in Modula-2 s.a. [Dal88]) oder spezielle Ereignisse (Interrupts) zugeteilt (s. Abb. 2.3). Beim Verlassen einer Coroutine wird ihr Zustand gesichert. Nach erneutem Transfer zu dieser verlassenen Coroutine wird in der Coroutine nach dem letzten Transfer fortgefahren. Dabei sind die Coroutinen unabhängig und nebenläufig.

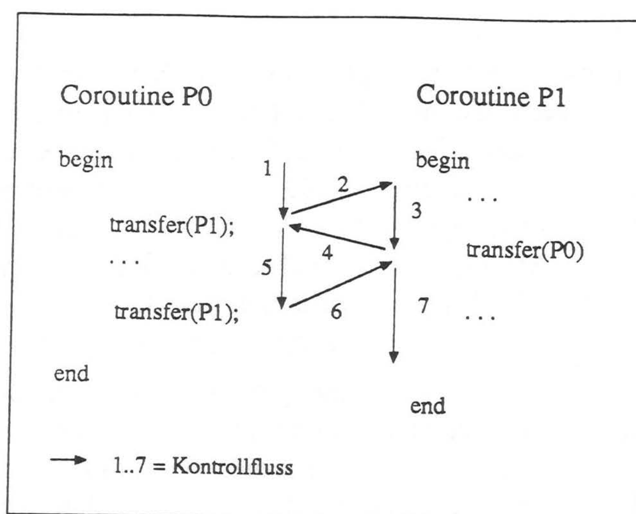

Abb. 2.3: coroutine/transfer

Diese explizite, programmierte Kontrollübergabe, ist auf einen Monoprozessor durch entsprechendes Setzen von Registern einfach realisierbar, bei Mehrprozessorsystemen ist dies jedoch so nicht möglich (vergl. [Los88]). Bei Kontrollübergabe in lose gekoppelten Mehrprozessorsystemen von einem Prozessor zum anderen müßte zuerst eine Kontrollübergabeanforderung vom Zielprozessor akzeptiert werden. Danach muß die auszuführende Coroutine in der gewünschten Umgebung aktiviert werden. Offen bleibt noch die Frage, was der die Kontrolle abgebende Prozessor nach der Kontrollübergabe macht.

2.2 Koordination paralleler Prozesse

Die Koordination paralleler Prozesse durch Synchronisation und Kommunikation kann über gemeinsam genutzte (*shared*) Variablen oder mittels Botschaften (*messages*) erfolgen [And83] .

2.2.1 Globale gemeinsame Variablen

Beim Zugriff auf gemeinsam genutzte Variablen ist ein gegenseitiger Ausschluß notwendig. Dafür muß entweder der Programmierer oder der Compiler sorgen. Der Programmier kann den gegenseitigen Ausschluß durch aktives Warten (busy waiting) implementieren. Die dazu notwendigen Algorithmen, die keine Deadlocks erzeugen dürfen, sind jedoch schwer zu verstehen [Ben82] und zu überprüfen.

Die Semaphoren [Dij68] mit den *P* und *V* -Operationen sind dazu besser geeignet. Sie müssen vom Programmierer jedoch ohne Unterstützung durch Programmierwerk-

zeuge korrekt verwendet werden.

Der Compiler kann die Benutzung globaler Variablen überprüfen, wenn die Zuordnung von globalen, gemeinsamen Variablen an Kodeabschnitte syntaktisch beschrieben wird. Jede gemeinsame Variable darf nur innerhalb eines Pfads benutzt werden. Dazu werden die gemeinsamen, globalen Variablen sogenannten *Resourcen*, eindeutig zugeordnet. Der gegenseitige Ausschluß wird dadurch gewährleistet, daß die überlappende Ausführung von Anweisungen in mehreren *bedingt kritischen Abschnitten* (*conditional critical regions*) [Bri72] auf Variablen aus derselben Resource nicht zugelassen wird. Dazu werden explizite Bedingungen (*guards* in CSP) verwendet.

Monitore sind zur Synchronisation besser geeignet als Semaphoren. In einem Monitor werden die sonst auf den gesamten Programmtext verstreuten Synchronisationsanweisungen textuell zusammengefaßt. Sowohl die Definition der beschreibenden Datenstruktur eines Betriebsmittels, als auch die darauf arbeitenden Prozeduren sind im Monitor zusammengefasst. Die internen Monitormechanismen (wait, awaited, signal) stellen sicher, daß sich jeweils nur ein Prozeß im kritischen Abschnitt befindet.

Das prozedurorientierte Monitor-Konzept basiert somit auf gemeinsamen Variablen. Objekte der auf dem Monitor-Konzept basierenden Sprache sind Prozesse und Monitore. Prozesse sind aktive Objekte, d.h. sie fordern andere Objekte auf etwas zu tun. Monitore (Module) sind passiv; sie erbringen auf ihren privaten Datenstrukturen eine Dienstleistung. Beispiele: Concurrent Pascal, Modula-2.

Dieses Modell bietet einen verständnismäßig leichten Übergang von Monoprozessorsystemen zu Multiprozessorsystemen. Es ist besonders geeignet bei gemeinsamem Speicher (*multiprocessing*). Bei globalem, gemeinsamem Speicher werden für den Datenzugriff nur Adressen verwendet. Bei verteilten Systemen müssen dagegen die entsprechenden Daten versandt werden. Daneben muß auf allen Prozessoren eine konsistente Datenhaltung gewährleistet werden, d.h. im Prinzip muß jeder Prozessor alle Daten erhalten. Die Simulation dieses Modells auf verteilten Systemen (*distributed processing*) ist daher aufwendig.

2.2.2 Botschaften

Die Prozessinteraktion wird beim botschaftsorientierten Sprachkonzept durch Datenaustausch mittels send/receive erreicht. Monitore sind in diesem Modell unbekannt. Es gibt keinen direkten Zugriff auf gemeinsame (globale) Daten. Aktive Objekte (Processes, Frames) beantworten Aufträge, die sie über Botschaften erhalten. Beispiele: CSP [Hoa78], PLITS (s.a. [Dil88]).

Dieses Modell ist besonders für lose gekoppelte Systeme gedacht. Es ist gut geeignet zur Bearbeitung verschiedener Daten auf mehreren Stufen (*pipelining*). Das dafür notwendige Netzwerk ist für den Benutzer transparent. Das System muß den Weg zum Empfänger bestimmen (routing) (s.a. [Dil88]).

Die Kommunikation zwischen verschiedenen Prozessen kann nach Art der Synchronisation und Art der Botschaft unterschieden werden [Neh88] .

Art der Synchronisation:

(1) *asynchron*: Eine Botschaft wird vom Sender an den Empfänger gesandt. Der Sender fährt sofort mit der nächsten Anweisung im Programm fort. Die Sendungen müssen gepuffert werden, da der Empfänger noch mit anderen Aufgaben beschäftigt sein kann.

(2) *synchron*: Sender und Empfänger warten aufeinander zum Datenaustausch.

Art der Botschaften:

(1) *Mitteilung*: In jedem Kommunikationsschritt wird nur eine Botschaft vom Sender an den Empfänger gesandt. Im asynchronen Fall ist dies der *no-wait-send-* im synchronen Fall der *Rendezvous- Mechanismus* (Ada).

(2) *Auftrag*: In jedem Kommunikationsschritt wird eine Auftragsnachricht vom Sender an den Empfänger gesandt und eine Ergebnisnachricht vom Empfänger an den Sender. Im asynchronen Fall ist dies der *Remote Service Invocation (RSI)-* im synchronen Fall der *Remote Procedure Call- Mechanismus.*

Ein beispielhaftes Zeitdiagramm für synchrone und asynchrone Mitteilungen folgt in Abb. 2.4:

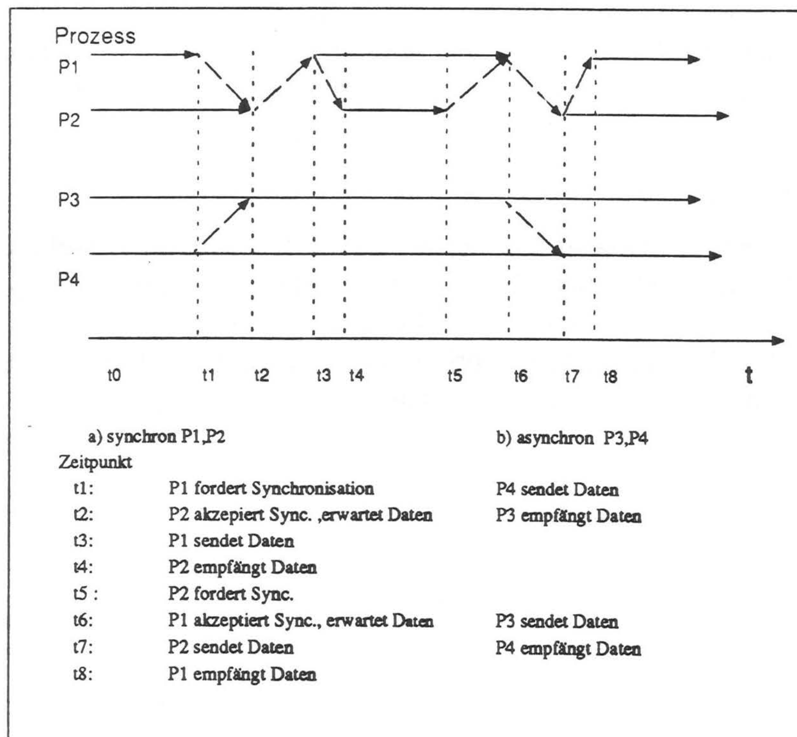

Abb. 2.4: Zeitdiagramme zum Botschaften senden

Zur Koordination durch Synchronisation und Kommunikation von lose gekoppelten Prozessen mittels Botschaften werden die elementaren Funktionen *send* und *receive*

benutzt. Die Aktion *send* sendet ein Datenpaket; die Aktion *receive* erwartet eines. Übertragen werden die Pakete entweder über Kanäle (CHAN in OCCAM) oder spezielle, globale Speicherbereiche (auch *Mailbox* oder *port* genannt).

Mit den besprochenen Koordinationsmitteln lassen sich dann die folgenden Protokollschemata implementieren.

Asynchrones Message Passing

Beim Botschaftenaustausch werden die Daten als Pakete über besondere Kanäle oder Speicherbereiche zwischen zwei Kommunikationspartnern ausgetauscht. Für jedes Paket gibt es eine Datenempfangsbestätigung.

Remote Procedure Call (*RPC*)

Die Parameter des Prozeduraufrufs werden als Botschaften an Stellvertreterprozesse auf anderen Prozessoren versandt. Sowohl der Ort der gewünschten Dienstleistung als auch der Weg dahin muß vom System bestimmt werden. Beim entfernten Prozeduraufruf treffen sich zwei Prozesse zur Kommunikation (Rendezvous). Auftraggeber (*client*) und Auftragnehmer (*server*) warten auf den jeweiligen Partner. Der Auftragnehmer wartet an seinem Synchronisationspunkt und übernimmt die Daten, bearbeitet sie und sendet die Ergebnisse an den darauf wartenden Auftraggeber zurück. Ein besonderes Problem stellt in diesem Fall das Erkennen der Terminierung von Prozessen dar.

Remote Service Invocation (*RSI*)

Bei der entfernten Dienstbeauftragung wird durch den Auftraggeberprozess (*client*) asynchron vom Auftragnehmerprozess (*server*) eine Dienstleistung angefordert. Der Auftragnehmer wartet nicht wie beim RPC auf das Ergebnis, sondern fährt in seiner Arbeit fort. Der Auftragnehmer erbringt die angeforderte Leistung und sendet das Ergebnis asynchron an den Auftraggeber zurück.

3. Parallelität für Attempto-2

Das Attempto-Konzept basiert auf Message-Passing. Da sich botschaftenorientierte Modelle im wesentlichen durch die Primitive *send* und *receive* realisieren lassen, wäre es am einfachsten, sie als Bibliotheksunterprogramme verfügbar zu machen. Der Programmierer soll diese Bibliotheksobjekte verwenden und selbst auf Konsistenz der Synchronisation und Kommunikation achten. Leider ist dieses Vorgehen bei großen Programmen zu fehleranfällig. Jeder Prozess kann im Prinzip mit jedem anderen, beliebigen Prozess kommunizieren; dabei geht die Übersicht aber schnell verloren. Deshalb entstand das im folgenden vorzustellende Parallelisierungskonzept.

3.1 Forderungen und Ziele

(1) Unser Ziel ist es, die Methoden zur Einführung von parallelen Sprachmitteln für Attempto-2 in eine vorhandene Sprache aufzuzeigen und die dafür notwendigen Werkzeuge zu erkennen, zu spezifizieren und zu entwickeln. Vom Programmierer erwarten wir nur die Spezifikation von parallel ausführbaren Folgen von Aktionen (Anweisungen).

(2) Ohne große Änderungen der bestehenden Programmierumgebung, soll während der Übersetzung eine möglichst weitgehende, maschinelle Überprüfung der Programme hinsichtlich der konsistenten Nutzung von parallelen Konstrukten erreicht werden. Automatisch generiert werden sollen die zusammengehörenden Aufrufe der Kommunikationsprimitive *send* und *receive* innerhalb eines parallelen Bereichs.

(3) Das Zurücksenden von Ergebnisdaten der verteilten Prozesse soll automatisch erfolgen. Der Empfang der Botschaften und die konsistente Änderung von globalen Daten durch die erhaltenen Ergebnisse soll dabei gewährleistet werden. Die mehrfache Änderung globaler Daten am Ende eines parallelen Bereichs wird als fehlerhaft erachtet. Dieser Fehler soll soweit möglich bereits während des Übersetzens angezeigt werden, spätestens aber zur Laufzeit zu einer Ausnahmebehandlung führen.

(4) Parallele Bereiche sollen unabhängig von der Anzahl der beteiligten Prozessoren (insbesondere auch bei einen Monoprozessorsystem) dasselbe Ergebnis erbringen.

3.2 Auswahl paralleler Sprachkonstrukte

Wir wollen, wie gesagt, möglichst einfach von vorhandenen sequentiellen Programmen zu parallelen Programmen kommen. Also muß von der Entwicklungsumgebung des Monoprozessorsystems der Übersetzer, Binder und Lader übernommen werden können, so daß nur für das Verteilen der Programme auf mehrere Prozessoren die notwendigen

Ergänzungen bzw. Erweiterungen (Präprozessor, Laufzeitunterprogramme, Konfigurator) neu geschaffen werden müssen.

Die Parallelität soll sprachlich folgendermassen beschrieben werden:

Anweisungen, die parallel ausführbar sind, werden textuell durch die Schlüsselworte **PARBEGIN** und **PAREND** geklammert. (Ähnlich dem PAR Konstrukt in Occam.) Als parallel auszuführende Anweisungen sollen nur Prozeduraufrufe erlaubt sein. Da parallel auszuführende Prozeduren auf verschiedenen Prozessoren ablaufen können, müssen solche Prozeduren nebenwirkungsfrei sein. Dies bedeutet, daß sie weder direkt noch indirekt, z.B. über von ihnen aufgerufenen Prozeduren, auf globale Variablen lesend oder schreibend zugreifen dürfen, sondern nur über ihre Werte und Variablenparameter mit dem aufrufenden Programm kommunizieren. Dies soll durch statische Analyse des Quellcodes sichergestellt werden. Die Nebenwirkungsfreiheit impliziert allerdings auch eine Gedächtnislosigkeit paralleler Prozeduren. Nur Prozeduren vom Typ des 'stateless servers' sind zugelassen. Dies ist eine sehr starke Einschränkung; es ist daher daran gedacht, den Zugriff auf globale Variablen in parallelen Prozeduren zuzulassen, wenn der Programmierer dies explizit fordert. Es bleibt dann allerdings seiner Verantwortung überlassen, wie er die Konsistenz globaler Daten auf verschiedenen Prozessoren gewährleistet.

Folgende Erweiterungen der Sprachmittel kommen in Betracht:

(1) *Parallel ausführbare Blöcke*
Die Klammern **PB** und **PE** umschließen Anweisungen (Pfad), die einen Block, bilden sollen. Anstelle von Prozeduraufrufen stehen dann Blöcke innerhalb des parallelen Bereichs. Dieses Konstrukt ist bequem bei der Programmerstellung, erhöht die Verständlichkeit und damit die Wartbarkeit eines Progamms. Es erfordert aber einen mächtigeren Präprozessor um dieselben Prüfungen (s.u.) zur Übersetzungszeit zu erhalten, wie sie bei Prozeduren und deren Aufruf erreicht werden.

(2) *Makros mit Variablen*
Damit sollen parallele Pfade mit denselben Anweisungen, aber anderen *konstanten* Werten einfach erzeugt werden, ohne sie explizit textuell zu kopieren. Leider werden die Anforderungen an den Präprozessor dabei wieder größer.

(3) *Pipe-Konstrukt*
Die Klammern **PipeBeg** und **PipeEnd** umschließen hintereinander ausgeführte Prozeduren, die eine Pipeline bilden. Die Prozeduren haben zwei Parameter für die Datenströme. Ein Parameter kennzeichnet den in die Verarbeitungseinheit einfliessenden Datenstrom, der andere den aus der Einheit hinausfliessenden Datenstrom. Weitere Parameter beschreiben Kontrolldatenströme zum Vorgänger und Nachfolger der Einheit. Damit sind die Datenpfade festgelegt und können bei Interprozessorkommunikation günstig auf schnelle Datenkanäle abgebildet werden.

Dieses Konstrukt ist sehr speziell und wird deshalb für Attempto vorerst nicht weiter betrachtet.

3.3 Randbedingungen und Designentscheidungen

Nach der Auswahl der Sprachkonstrukte sollen nun die prinzipiellen Ideen der Verwirklichung von Parallelität in Attempto skizziert werden. Details findet man in der Beschreibung der Erweiterungen der Attempto-2 Laufzeitbibliothek [Lut89] .

Mit dem folgenden, kleinen Beispiel wollen wir den Übergang von der Quelle bis zur Ausführung veranschaulichen. Wir verwenden dazu ein PARBEGIN/PAREND- Konstrukt.
Unser Beispiel:

```
VAR x,y,z,w: INTEGER;
  ....
PROCEDURE P1(VAR PerReference: INTEGER);
VAR local: INTEGER;
BEGIN
  ....
END P1;

  ....

PARBEGIN
    P1(x);
    P2(y);
    P3(z);
    P1(w);
PAREND
  ....
```

Abb. 3.1: Paralleler Bereich

Hier sollen die Prozeduren P1,P2,P3 parallel ausgeführt werden. Dabei wird P1 mit zwei verschiedenen Parameterwerten aufgerufen.

Wir haben dabei folgendes Bild vor Augen (s. Abb. 3.2):

(Nebeneinander angeordnet sind der Quelltext, die entsprechenden Kommunikations-, Synchronisationsaufrufe und der entsprechende schematische Ablauf. Auf gleicher Höhe befinden sich einander entsprechende Beschreibungen.)
Im Programm PR ist im Quelltext eine PARBEGIN,PAREND-Klammer vorhanden, die angibt, dass die Prozeduren P1,P2,P3 und P1 mit den Parametern x,y,z und w parallel auf verschiedenen Prozessoren ablaufen könnten. Im Bild sind maximal vier parallele Pfade möglich. Im Idealfall (bei 5 vorhandenen Prozessoren), würde also P1,P2,P3,P1 auf den Prozessoren (Auftragnehmer) AN(i) i=1..4 ablaufen, nachdem der Auftrageberprozessor AG, der den vorangehenden sequentiellen Teil ausführt, die notwendigen Daten verteilt hat. Nach dem Verteilen wartet der Auftraggeber und sammelt dann bei PAREND die Ergebnisse der Auftragnehmer ein.

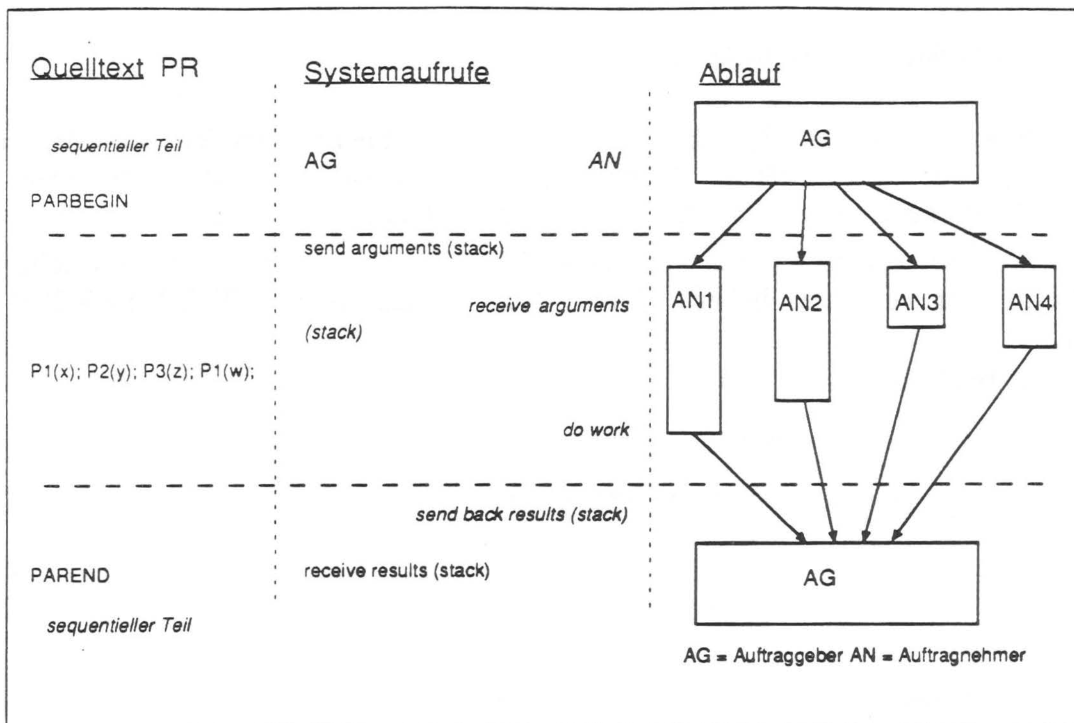

Abb. 3.2: Kontroll- und Datenfluß

Konzept in Attempto-2

Unsere Implementierung dieser Vorstellung beruht auf folgenden Ideen:

(1) PARBEGIN wird von einem Präprozessor umgesetzt in einen Aufruf eines Laufzeitunterprogramms ParBegin(...) mit Parametern, die jeden parallelen Bereich innerhalb eines Programmablaufs eindeutig kennzeichnen. Das Laufzeitunterprogramm besorgt die Information über die aktuelle Bindung der Tasks an die Prozessoren [Lut89] .

(2) PAREND wird umgesetzt in den Aufruf ParEnd(...). Die Aufgabe von ParEnd ist es, die Ergebnisse der einzelnen Aktionen (parallelen Pfade) zu sammeln und zu prüfen.

(3) In Prozeduren eines parallelen Pfads dürfen nur explizit als Referenzparameter übergebene globale Variablen benutzt werden.

(4) Beim Aufruf einer parallel abzuarbeitenden Prozedur wird zusätzlich zur Adresse eines Referenzparameters (VAR- Parameter) ein Werteparameter mit dem aktuellen Wert des Referenzparameters und die Typgröße (TSIZE) dieses Wertes übergeben. Im Auftragnehmer wird dann auf dem Werteparameter gearbeitet (s.a. [Lut89]) der dann auch zurückgesandt wird. Typgröße und Adresse werden im

Auftraggeber von ParEnd ausgewertet und zur Sicherung der Datenkonsistenz beim Aktualisieren der globalen Variablen durch die Ergebnisse der Aufragnehmer beachtet.

(5) Die Tasks werden von der Programmierumgebung (Präprozessor) so modifiziert, daß die daraus erzeugten Prozesse nach dem Aufruf auf Grund der aktuellen Verteilung entweder die Aufgabe des Auftraggebers, oder die eines Auftragnehmers übernehmen können.

(6) Jede Aktion, d.h. Ausführung einer Task lokal oder auf einem anderen Prozessor, wird angestoßen durch eine Remote Service Invocation (s.a. 2.2.2).

Ersetzungen wie in den Punkten (1),(2),(5) werden dabei vom Präprozessor vorgenommen (textuelle Ersetzung), oder dadurch, daß Objekte der Bibliotheken durch neue Objekte mit demselben Namen aber erweiterter Funktion ersetzt werden.

Vorstellbar sind also folgende Alternativen:

(1) geeignete Laufzeitunterprogramme wie *procentry* [Tan83] werden in der Laufzeitbibliothek durch erweiterte ersetzt.

(2) explizites Einfügen von zusätzlichen Anweisungen durch den Präprozessor.

Hierbei ist die Designentscheidung zu treffen, ob die im Auftraggeber und Auftragnehmer notwendigen Aufrufe zum Senden, Empfangen und zum Einordnen der Resultate im expandierten Text sichtbar vom Präprozessor eingefügt werden, oder ob die Leistung innerhalb der Systemunterprogramme erbracht werden soll. Wenn ein Compiler vor einem Prozeduraufruf keine Aufrufe für besondere Bibliotheksobjekte erzeugt, oder die erforderliche Leistung nicht in diesem Aufruf erbracht werden kann, dann kommt nur die Expansion durch den Präprozessor in Frage. Die Expansion des Textes ist auch im Fehlerfall von Vorteil, da die Kommunikation dem Programmierer sichtbar ist. Durch die Ausführlichkeit wird aber ein expandiertes Programm unübersichtlich.

Die Ersetzungen enthalten die notwendige Synchronisation und Kommunikation sowie die Aufrufe von geeigneten Prozeduren zur Überprüfung der Konsistenzanforderungen während der Laufzeit. Konsistent müssen im parallelen Bereich sein: die Benutzung globaler Variablen, der Botschaftenversand an die Tasks und der Empfang der Ergebnisse. Dazu müssen alle Erweiterungen beitragen.

Beschränken wir uns auf die Kennzeichnung von parallel ausführbaren Prozeduren die nur zum Hauptprogramm lokal sind, dann behaupten wir, daß die explizite Parallelität durch Präprozessor, unterstützende Laufzeitbibliothek [Lut89] und Konfigurator erreicht werden kann, ohne daß eine neue Programmiersprache zu implementieren ist, und daß vorausgesetzt werden kann, daß der Programmierer keine Kenntnisse der Implementierung der Parallelitätskonstrukte benutzt oder benötigt.

4. Programmgenerierung

Nun beschreiben wir die Auswirkungen des Konzepts auf die Programmgenerierung. Zuerst betrachten wir die Veränderungen beim Schreiben und Übersetzen eines Programms. Danach erörtern wir die Bedeutung und Realisierung der Parameterübergabe innerhalb eines Programms. Anschließend beleuchten wir die zusätzlich erforderlichen Laufzeitunterprogramme. Hinterher formulieren wir die daraus entstehenden zusätzliche Anforderungen an die Standardbibliothek. Abschließend verfolgen wir das Linken, Verteilen (*Schedule*) und Zuteilen (*Dispatching*) der Tasks an die Prozessoren.

4.1 Sequentielle Programme

Bei einer typischen Programmgenerierung für sequentielle Systeme haben wir folgenden Ablauf:

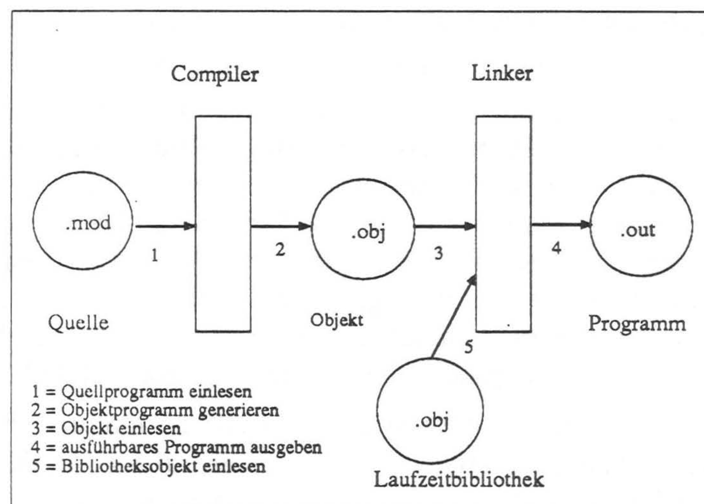

Abb. 4.1: Klassische Programmcodegenerierung

Zur Kennzeichnung eines speziellen Typs von Information sollen bei den Dateinamen die sogenannten Dateiextensions dienen. Bei vielen Betriebssystemen beginnt eine *Extension* des Dateinamens mit dem Trennzeichen '.' , gefolgt von drei alphanumerischen Zeichen.

Ein Quellprogrammm mit der Extension *.mod* wird vom **Compiler** in ein Objekt (*.obj*) übersetzt. Danach wird dieses Objekt mit anderen Objekten aus den Bibliotheken vom **Linker** zu einem ausführbaren Programm (*.out*) zusammengebunden. Dieses Programm wird dann beim Aufruf vom **Systemlader** zu einem Prozess gemacht und erhält vom **Systemdispatcher** den Prozessor zugeteilt.

4.2 Parallele Programme

Parallele Programme setzen sich aus Tasks zusammen. Wenn diese verteilt ablaufen sollen, dann müssen Werkzeuge wie Präprozessor, Profiler, Konfigurator und die Laufzeitunterprogramme zumindest folgende Aufgaben erledigen:

(1) Parallel ausführbare Programmteile kennzeichnen und die parallele Bearbeitung vorbereiten (send/receive- Aufrufe einfügen).

(2) Eine parallele/sequentielle Ablaufstruktur (Parallel-Schedule) ermitteln.

(3) Programmteile und Daten den ausführenden Recheneinheiten zuordnen (Parallel-Dispatching) .

(4) Den parallelen Ablauf zur Laufzeit unterstützen.

Bei uns sollen parallele Programme in der modulorientierten Programmiersprache Modula-2 erstellt werden. Am Besten unterstützt unser Vorhaben eine Laufzeitunterprogramm-Bibliothek mit modifizierten Unterprogrammen und zusätzlichen Kommunikationsprimitiven. Die Erfahrung mit, bei der expliziten Benutzung von Semaphoren vergessenen P oder V- Aufrufen läßt eine maschinelle Generierung von Anweisungen zur Parallelisierung wünschenswert erscheinen. Deshalb sollen diese systematisch durch einen Präprozessors erzeugt werden.

Der **Präprozessor** erstellt aus einer parallelen Quelle (mit der Extension *.par*) ein übersetzbares *normales* Quellprogramm (*.mod*). Daneben generiert er die Quellcode-Dateien *.cnf.def* und *.cnf.mod*, welche die Tabellen des Konfigurators beschreiben. In diesen Moduln werden auch die Funktionen vereinbart, welche die entsprechenden Tabellen zur Laufzeit füllen. Alle Tabellenzuweisungen als Quelltext für *.cnf.mod* zu erzeugen erschien uns zu starr, zu aufwendig und zu langsam. Die explizite statische Parallelisierungsinformation wird entsprechend den Tabellen binär in eine zusätzliche Datei *.cfd (configuration data)* ausgegeben.

Beim Übersetzen durch den **Compiler** wird ein *Profil* [Gra82] vorbereitet werden. Durch einen sequentiellen Programmablauf (*.out*) wird dann ein Laufzeitprofil erstellt. Das Ergebnis wird vom **Profiler** aufbereitet und in einer Datei (*Profil-Information .prf*) für den **Konfigurator** gesichert. Dazu ist eine *profilierende* Version der normalen Laufzeitbibliothek notwendig!

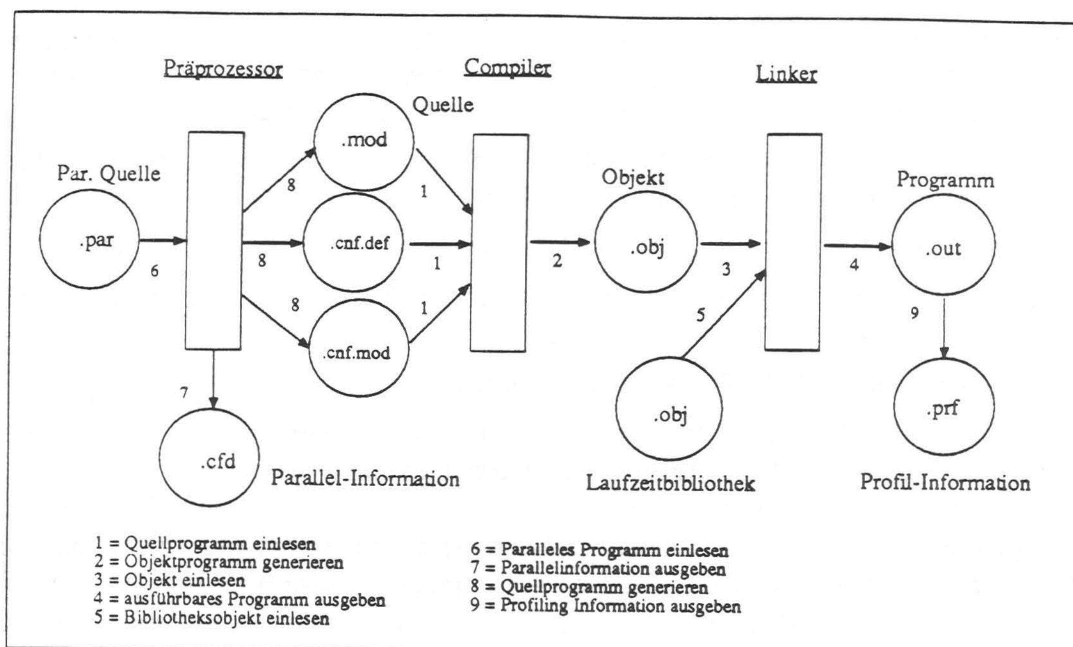

Abb. 4.2: Codegenerierung für verteilte Programme

Der Konfigurator erstellt aus der statischen Parallelstrukturbeschreibung *.cfd* und der Information *.prf* aus einem Profilierungslauf die Schedule-Information für die Tasks. Diese beschreibt die Anforderungen an die Zielprozessoren und deren Kommunikationsverbindungen für das zu ladende Programm.

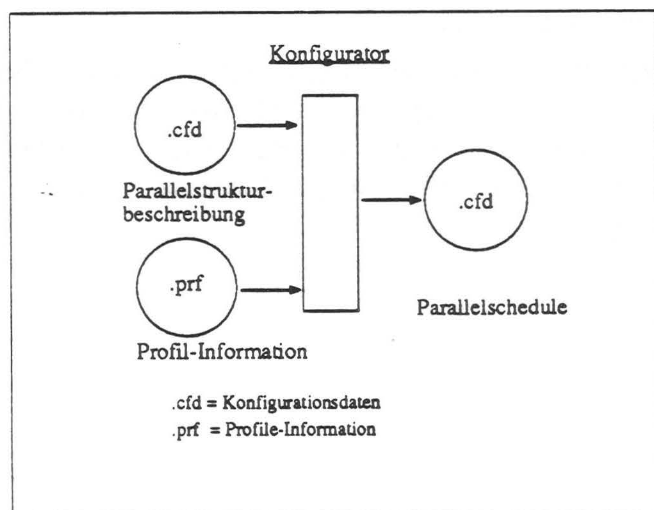

Abb. 4.3: Scheduling verteilter Programme

Der Systemlader lädt dann den initialen Prozess. Dieser Prozess liest sich die binären Werte der Konfiguration aus *.cfd* ein. Zusätzlich besorgt er die Beschreibung der

verfügbaren Prozessoren (activeProcessors). Der initiale Prozess startet die weiteren Prozesse und übergibt ihnen die zur Laufzeit aktuelle Konfigurationsinformation.

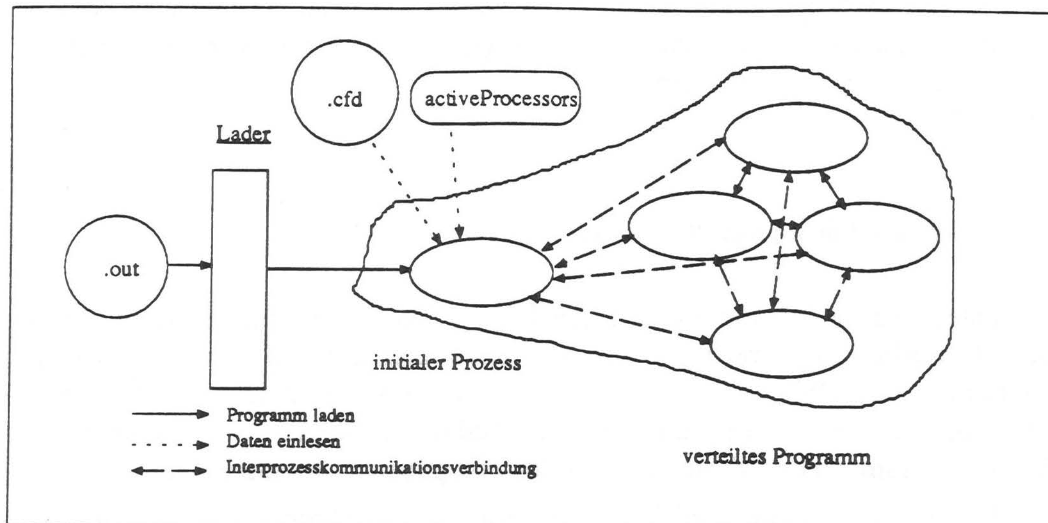

Abb. 4.4: Dispatching verteilter Programme

Damit ist das statische Wissen zur Programmstruktur ergänzt um die aktuelle dynamische Dispatching-Information, für die kommunikationsunterstützenden Laufzeitroutinen verfügbar. Am Besten werden alle relevanten Informationen in einer Datenbank gespeichert, damit auch die Diagnose (Fehlertoleranz) darauf zugreifen kann.

5. Taskanstoß

Nach dem Laden und Initialisieren des Prozessystems auf den Prozessoren beginnt der Ablauf des parallelen Programms. Im folgenden betachten wir den Anstoß der Tasks und ihre Versorgung mit Daten.

5.1 Prozeduraufruf in sequentiellen Programmen

Betrachten wir zuerst die Datenübergabe an Prozeduren im sequentiellen Falle. Für einen Prozeduraufruf werden zuerst die Parameter auf den Stack ge*push*ed, dann der Programmzähler (PC) auf den Stack gelegt (Rückkehradresse) und letztlich der PC mit der Adresse der auszuführenden Prozedur geladen. Bei Variablenparametern (Referenzparameter) wird die Adresse der übergebenen Variablen auf den Stack gelegt.

Im Unterprogramm wird i.d.R. als erstes der Stackpointer für den Parameterzugriff gesichert. Dann wird der Platz für die prozedurlokalen Variablen reserviert. Am Prozedurende wird der Stackpointer korrigiert und die Rückkehradresse in den PC geladen.

Nach dem Aufruf und der lokalen Speicherreservierung sieht der Stack folgendermassen aus (kleinere Adressen oben):

```
SP -->    local   (* lokale Variable    *)
a0 -->    PC      (* Rueckkehradresse   *)
          x       (* Prozedurparameter  *)
```

Abb. 5.1: Stackaufbau

Die Daten auf dem Stack werden meist relativ zum Zeiger (basepointer) auf die Rückkehradresse adressiert. Die Parameter werden in der Reihenfolge des Vorkommens im Aufruf auf den Stack geschrieben. Prozedurlokale Variable werden auf dem Stack angelegt.

5.2 Remote Procedure Call

Beim Remote Procedure Call- (RPC) Mechanismus werden die Parameter in eine Message verpackt und an einen Stellvertreterprozess auf einem Prozessor gesendet (send) und dann auf das Ergebnis gewartet (receive). Der externe Prozessor erbringt die angeforderte Dienstleistung und sendet das Ergebnis zurück. Im client/server- Modell von verteilten Systemen wird dieser Mechanismus gern verwendet.

Ein Beispiel:

```
RPC(servername,servicename,params,results);
```

Abb. 5.2: Remote Procedure Call

dies wird expandiert zu:

```
local (client)                    remote (server)
---------------                   -----------------
send(server,service,params);
                                  receive(service,params);
(* wait *)                        (* do service *)
                                  send(client,results);
receive(results)
```

Abb. 5.3: Expandierter Remote Procedure Call

Verwendet wird dieser Mechanismus z.B. in der Newcastle-Connection [Nee82] . In einer vom benutzenden Programm aus transparenten Schicht werden Systemcalls, die an andere (remote) Maschinen gehen sollen, herausgefiltert und mit dem RPC an dieselbe transparente Schicht auf den remote Maschinen übergeben. Dort werden die Systemaufrufe an das lokale Betriebssystem gerichtet und die Ergebnisse zurückgesandt.

5.3 Remote Service in Attempto-2

In unserem Fall soll für den Taskanstoß der Stackmechanismus mit dem Send/Receive-Mechanismus verknüpft werden. Dies entspricht einem **copy, restore**- Mechanismus [Aho86], neu angewandt auf parallele, verteilte Programme. Da nach der Expandierung Parameter nur per Wert übergeben werden, betrachten wir den für diesen Aufruf belegten Stackbereich als Inhalt (Nachricht) einer zu versendenden Botschaft. Im Unterschied zu inhomogenen Systemen, bei denen die übergebenen Werte prozessortyp- und betriebssystem- unabhängig beschrieben und in Botschaften gepackt werden müssen, können sie im homogenen Multiprozessorsystem als linearer, zusammenhängender Adressbereich interpretiert werden. Sind die Prozessoren in einem Multiprozessorsystem eng gekoppelt, dann genügt es, dem Empfänger nur die Grenzen des Stacks mitzuteilen. Dann kann sich der Empfänger die Daten aus dem gemeinsamen globalen Speicherbereich per *Direct Memory Access* DMA (Quelle,Länge,Ziel) in seinen lokalen Speicher holen.

Benutzen wir die Kopien des gesamten Textteils eines Programms auf jedem Prozessor, dann stimmen sogar die relativen, prozesslokalen Aufrufadressen der Prozeduren überein [Kar87],[Zha86] . Eine Kopie des gesamten Programms auf einem Prozessor des homogenen Systems soll *Einhüllender Prozess (E-Prozess)* genannt werden.

Sind beim Taskanstoß VAR- Parameter vorhanden, dann müssen sie wertmäßig in die Botschaft kopiert werden. Wird ein Variablenwert verändert, dann muß dieser im Ergebnisstack zurückgegeben werden. Im allgemeinen müssen die Werte aller Variablen zurückgesandt werden, insbesondere auch Felder, deren Elemente nur gelesen und nicht verändert werden. Ein Compiler mit Datenflußanalyse zur Codeoptimierung [Aho86] könnte diese Datenmenge reduzieren, da er die Variablen bestimmen kann deren Wert sich ändert.

Das Übergeben der Daten vom Auftraggeber an die Task im Auftragnehmer erfolgt auf dem ATTEMPTO-System in drei Schritten.

(1) Schreiben in den Port [Dal87] und damit versenden.

(2) Lesen im E-Prozess.

(3) Lesen in der Task, wie bei einem Prozeduraufruf üblich.

Vor dem Ende einer Aktion werden die Ergebnisse dann an den Auftraggeber verschickt. Dabei benutzen wir, daß Wertparameter auf dem Stack angelegt werden und Zuweisungen diesen Stackbereich ändern und senden einfach den Stack zurück. Beim Auftraggeber müssen die Werte der entsprechenden Variablen geändert werden. Details finden sich in der Beschreibung zur Attempto-2 Laufzeitbibliothek [Lut89].

6. Schlußbetrachtung

Aus den einfachen Anforderungen, die wir am Anfang dieses Berichts erwähnt haben resultieren verzwickte, problematische Folgeforderungen. Dennoch meinen wir ein verwirklichbares Konzept aufgezeigt zu haben, mit dessen Hilfe sequentielle, blockorientierte, prozedurale Programme auf Blockebene aus Anwendersicht einfach parallelisiert werden können.

Die *einfache* Parallelität auf Prozedurebene wird in unserem Falle nicht durch eine neue Programmiersprache realisiert, sondern stattdessen durch eine Kombination von Präprozessor, Konfigurator, Dispatcher und Laufzeitbibliothek erreicht, was die Übertragung auf andere Sprachen und die Portierung auf andere Systeme erleichtern dürfte.

Literatur

[Aho86] Aho,A.V.;Sethi,R.;Ullman,J.D. "Compilers Principles, Techniques, and Tools" Addison-Wesley Publishing Company, 1986

[And83] Andrews,G.R;Schneider,F.B "Concepts and Notations for Concurrent Programming"
Computing Surveys, Vol. 15, #1 (March 1983), pp.3-83

[Ben82] Ben-Ari,M. "Principles of Concurrent Programming"
Prentice Hall 1982

[Bri72] Brinch Hansen,P. "Structured multiprogramming"
CACM, Vol.15, #7, (July 1972), pp.574-578

[Bri75] Brinch Hansen,P. "The Programming Language Concurrent Pascal"
IEEE Trans. on Softw. Eng., SE-1, #2 1975, pp.199-207

[Dal87] Dal Cin,M.;Brause,R.;Lutz,J.;Dilger,E.;Risse,Th. "ATTEMPTO: An Experimental Fault-Tolerant Multiprocessor System"
Microprocessing and Microprogramming 20(1987) pp.301-308, North Holland

[Dal88] Dal Cin,M.;Lutz,J.;Risse,Th. "Programmierung in Modula-2"
3.Aufl., B.G. Teubner Stuttgart 1988

[Den66] Dennis,J.B.;van Horn,E.C "Programming semantics for multiprogrammed computations"
CACM Vol.9, #3 (March 1966), pp. 143-155

[Dij68] Dijkstra,E.W. "Cooperating Sequential Processes"
In "*Programming Languages*"; (ed.) F. Genuys
Academic Press New York, 1968, pp.43-112

[Dil88] Dilger-Klett,R. "Eine Modula-2-PLITS Implementierung für Lokale Netze"
Diplomarbeit am Fachbereich Informatik VSFT, J.W. Goethe Universität Frankfurt, 1988

[Gra82] Graham S. L; Kessler P.B; McKusick M.K. "gprof: a Call Graph Execution Profiler"
SIGPLAN Notices Vol. 17, #6 (June 1982), pp.120-126

[Hoa78] Hoare,C.A.R. "Communicating Sequential Processes"
CACM, Vol. 21, #8 (Aug. 1978), pp.666-677

[Kar87] Karp,A.H. "Programming for Parallelism"
IEEE Computer, (May 1987), pp.43-57

[Los88] Loske,U. "Implementierung von Remote Coroutinen unter Modula-2"
Diplomarbeit am Fachbereich Informatik VSFT, J.W. Goethe Universität Frankfurt, 1988

[Lut89] Lutz,J.;Brause,R.;Dal Cin,M.;Philipp,Th.
"Die Erweiterungen der ATTEMPTO-2 Laufzeitbibliothek"
Interner Bericht 2/89, Fachbereich Informatik Universität Frankfurt

[May83] May,D. "Occam"
ACM Sigplan Notices, Vol. 18,#8 (April 1983), pp.69-79

[Nee82] Needham,R.M.; Herbert,A.J. "The Cambridge Distributed Computing System"
Addison-Wesley 1982

[Neh88] Nehmer,J. "Entwurfskonzepte für verteilte Systeme - eine kritische Bestandsaufnahme"
IFB 187, GI- Jahrestagung 1988, pp. 70-96

[Tan83] Tanenbaum,A.S et al. "A Practical Tool Kit for Making Portable Compilers."
CACM, Vol.26, #9 (Sep. 1983), pp. 654-660

[Uni79] Unix "Unix Programmer's Manual, section 1"
Bell Laboratories, Murray Hill, NJ. Jan. 1979

[Zha86] Zhang,D;Lu,M. "Process Name Resolution in Fault-Tolerant CSP Programs"
ACM Op. Sys. Rev. Vol.20, #4 (Oct 1986), pp.9-15

bisher erschienen:

1/81	Spaniol, Otto:	Performance evaluation of the "virtual subchannel concept" for satellite link communication
2/81	Spaniol, Otto:	Analysis and performance evaluation of DBMS-model
3/81	Wotschke, Detlef Goldstine, Y. Price, Y.K.:	A pushdown automation or a context-free grammar - which is more economical?
4/81	Wotschke, Detlef Goldstine, Y. Price, Y.K.:	On reducing the number of states in a pda
5/81	Kemp, Rainer:	The average height of planted plane trees with m leaves
1/82	Meyer auf der Heide, F.:	Efficiency of universal parallel computers
2/82	Spaniol, Otto Hunsmann, Norbert Nawrath, Franz:	Leistungsbewertung von Rechnersystemen am Beispiel eines Rechnerverbundsystems
3/82	Wegener, Ingo:	The discrete search problem and the construction of optimal allocations
4/82	Wegener, Ingo:	Best possible asymptotic bounds on the depth of monotone functions in multivalued logic
5/82	Meyer auf der Heide, F.:	Infinite cube-connected cycles
6/82	Wegener, Ingo:	Relating monotone formula size and depth of Boolean functions
7/82	Geihs, Kurt:	Analytische und simulative Betrachtung eines Multiplexors mit time-out
8/82	Meyer auf der Heide, F.:	A polynominal linear search algorithm for the n-dimensional Knapsack problem
9/82	Wegener, Ingo:	On the complexity of discrete search problem with positive switch cost
1/83	Jaschinski, Jörg Wegener, Ingo:	Optimal nonadaptive strategies for the search in detection functions
2/83	Meyer auf der Heide, F.:	Efficient simulations among several models of parallel computers

3/83	Wegener, Ingo:	Optimal decision trees and one-time only branching programs for symmetric Boolean functions
4/83	Wegener, Ingo:	Optimal search with positive switch costdecidability, complexity, problems
5/83	Trum, Peter Wotschke, Detlef:	Descriptional complexity for programs
6/83	Meyer auf der Heide, F.:	Lower time bounds for integer programming with two variables
7/83	Wegener, Ingo:	On the complexity of slice functions
1/84	Linnemann, Volker:	Deterministic processor scheduling with communication costs
2/84	Linnemann, Volker:	Grammar rules: a new formalism for defining data types
3/84	Linnemann, Volker:	On the relationship between context free grammars and hierachical data structures
4/84	Jarke, Matthias Koch, Jürgen:	Query optimization in database systems
5/84	Wegener, Ingo:	On the complexity of branching programs and decision trees for Clique functions
6/84	Meyer auf der Heide, F.:	Lower bounds for solving linear diophantine equations on Random Access Machines
7/84	Meyer auf der Heide, F.:	Lower time bounds for testing the solvability of diophantine equations on several parallel computational methods
8/84	Wegener, Ingo:	Time-space trade-offs for branching programs
9/84	Wegener, Ingo:	An improved complexity hierarchy on the Boolean network complexity and formula size of Boolean functions
10/84	Wegener, Ingo:	The critical complexity of all(monotone) Boolean functions and monotone graph properties
1/85	Kintala, Chandra M.R. Wotschke, Detlef:	Concurrent conciseness of degree, probabilistic, nondeterministic and deterministic finite automata
2/85	Seidl, Helmut:	A quadratic regularity test for non-deleting

		macro s grammars
3/85	Wegener, Ingo:	More on the complexity of slice functions
4/85	Fellmann, Annemarie:	Optimal algorithms for the multiplication in simply generated local algebras
1/86	Bublitz, Siegfried Schürfeld, Ute Voigt, Bernd Wegener, Ingo:	Properties of complexity measures for prams and wrams
2/86	Kemp, Rainer:	The analysis of an additive weight of random trees
3/86	Brustmann, Bettina Wegener, Ingo:	The complexity of symmetric functions in bounded-depth circuits
4/86	Risse, Thomas:	On the symbolical evaluation of the reliability of systems whose structure function is given by any Boolean expression in its components
5/86	Dal Cin, Mario Brause, Rüdiger Lutz, Joachim Dilger, Elmar Risse, Thomas:	ATTEMPTO: An experimental fault-tolerant multiprocessor system
6/86	Wegener, Ingo:	The range of new lower bound techniques for WRAMs and bounded depth circuits
7/86	Dal Cin, Mario:	Ein Diagnoseverfahren für Verteilte Systeme
1/87	Risse, Thomas:	On the number of multiplications needed to evaluate the reliability of k-out-of-n systems Modelling interrupt based interprocessor communication by Time Petri Nets
2/87	Roll, Georg Strugala, Michael Tavangarian, Djamshid Waldschmidt, Klaus:	Ein Assoziativprozessor auf der Basis eines modularen vollparallelen Assoziativspeicherfeldes
3/87	Waldschmidt, Klaus Roll, Georg:	Entwicklung von modularen Betriebssystemkernen für das ASSKO-Multi-Mikroprozessorsystem

4/87		Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen ; 3.2.1987, Universität Frankfurt/Main
5/87	Seidl, Helmut:	Parameter-reduction of higher level grammars
6/87	Kemp, Rainer:	On systems of additive weights of trees
7/87	Kemp, Rainer:	Further results on leftist trees
8/87	Seidl, Helmut:	The construction of minimal models
9/87	Weber, Andreas Seidl, Helmut:	On finitely generated monoids of matrices with entries in N
10/87	Seidl, Helmut:	Ambiguity for finite tree automata
1/88	Weber, Andreas:	A decomposition theorem for finite-valued transducers and an application to the equivalence problem
2/88	Roth, Peter:	A note on word chains and regular languages
3/88	Kemp, Rainer:	Binary search trees for d-dimensional keys
4/88	Dal Cin, Mario:	On explicit fault-tolerant, parallel programming
5/88	Mayr, Ernst W.:	Parallel approximation algorithms
6/88	Mayr, Ernst W.:	Membership in polynomial ideals over Q is exponential space complete
1/89	Lutz, Joachim [u.a.]:	Parallelisierungskonzept für ATTEMPTO-2