FACHBEREICH INFORMATIK

Universität Frankfurt

DIE ERWEITERUNGEN DER ATTEMPTO-2

LAUFZEITBIBLIOTHEK

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

INTERNER BERICHT 2/89

Die Erweiterungen der ATTEMPTO-2 Laufzeitbibliothek

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

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

Abstrakt

Dieser Bericht beschreibt die speziellen Aspekte einer Laufzeitbibliothek zur Parallelisierung und Verteilung von Programmen im homogenen, nachrichtengekoppelten Mehrrechnersystem ATTEMPTO-2, das zur Untersuchungen von parallelen Systemen und Fehlertoleranz dient.

Inhaltsverzeichnis

1. Einleitung

1.1. Ziele

Im Bericht "Parallelisierungskonzept von ATTEMPTO-2" [Lu89a] beschrieben wir das Konzept einer Softwareumgebung für die Parallelprogrammierung von Mehrrechnersystemen. Ein wesentlicher Aspekt des Konzepts ist, daß anstelle einer Modifikation existierender Compiler die parallelen Konstrukte durch einen Präprozessor und eine unterstützende Laufzeitbibliothek implementiert werden.

Im folgenden gehen wir näher auf die Voraussetzungen für diese Laufzeitbibliothek ein. Zugleich geben wir konkrete Realisierungshinweise. Zur Laufzeitbibliothek unserer Entwicklungssprache Modula-2 zählen die Standardunterprogramme, die in den Modula-2 Standardmoduln [DLR88] (z.B InOut, Storage usw.) vorhandenen sind. Für ATTEMPTO-2 wird sie um die zur Parallelisierung notwendigen zusätzlichen Prozeduren erweitert, die als *P-Laufzeitprozeduren* bezeichnet werden sollen.

1.2. Begriffe

Als erstes seien, wie auch im "Parallelisierungskonzept von ATTEMPTO-2", im folgenden verwendete Begriffe eingeführt:

Parallele Bereiche: Programmabschnitte, in denen sich der Kontrollfluß des Programms in mehrere *parallele Pfade* aufteilen kann.
Task: Paralleler Pfad, der durch eine Prozedur beschrieben wird. Der Aufruf einer solchen Prozedur wird *Anstoß* zur Ausführung einer Task genannt.
Aktion: Ausführung einer Task.
Aktionsablaufplan: Plan zur Verteilung der Aktionen auf ein Mehrrechnersystem.
Prozeß: Objekt, dem vom Betriebssystem ein Prozessor zugeteilt werden kann.
Einhüllender-Prozeß (E-Prozeß): Kopie des gesamten, ablauffähigen und zu verteilenden Programms.
E-Prozeß-Cluster: Gesamtheit der vom System zur Ausführung des verteilten Programms geladenen E-Prozesse.
Virtueller Prozeß: Folge von Prozessen (Aktionen), deren Ausführung vom Aktionsablaufplan einem E-Prozeß zugeordnet wird.
Taskverteilung: Zuordnen eines Virtuellen Prozesses zu einem E-Prozeß.
Auftraggeber: Ein E-Prozeß, der zur Laufzeit Aufträge zur Ausführung von Tasks an andere E-Prozesse, den *Auftragnehmern* des E-Prozeß-Clusters verteilt.

2. Parallele Bereiche

In einem parallelen Bereich sind die aufgerufenen Prozeduren (Tasks), *parallel* ausführbar. Ob diese Aktionen *lokal* oder *entfernt (remote)* ausgeführt werden, wird zur Laufzeit bestimmt. Dazu werden vom Laufzeitsystem Tabellen verwaltet, die von einem Konfigurator (s. [Lu89a]) als Aktionsablaufplan erzeugt werden und von der Auftragsverteilung entsprechend den aktuellen Systemgegebenheiten benutzt werden (s.a. Kap. 5 über "Konfigurierung und Verteilung eines parallelen Programms"). Im folgenden skizzieren wir, welche Besonderheiten dabei beachtet werden müssen.

Als Ausgangspunkt betrachten wir das Bild einer möglichen parallelen Konfiguration (Abb. 2.1). Die Ellipsen stellen E-Prozesse auf Prozessoren dar. Die Kreise stellen auszuführende Teilaufgaben (Tasks) des Gesamtprogramms dar. Die Pfeile kennzeichnen Kommunikationsverbindungen.

Zur Notation: Im folgenden sollen die Prozeduren eines parallelen Bereichs durch Prozedurnamen mit Kleinbuchstaben beschrieben werden, eine Aktion dagegen mittels Großbuchstaben. PARBEGIN und PAREND begrenzen den parallelen Bereich. Die Anweisungen p1(x),p2(y,s),p3(z),p1(w) stehen für den Aufruf von Prozeduren (Taskanstoß). P1(x),P2(y,s),P3(z),P1(w) stehen für die entsprechenden Aktionen (Ausführungen).

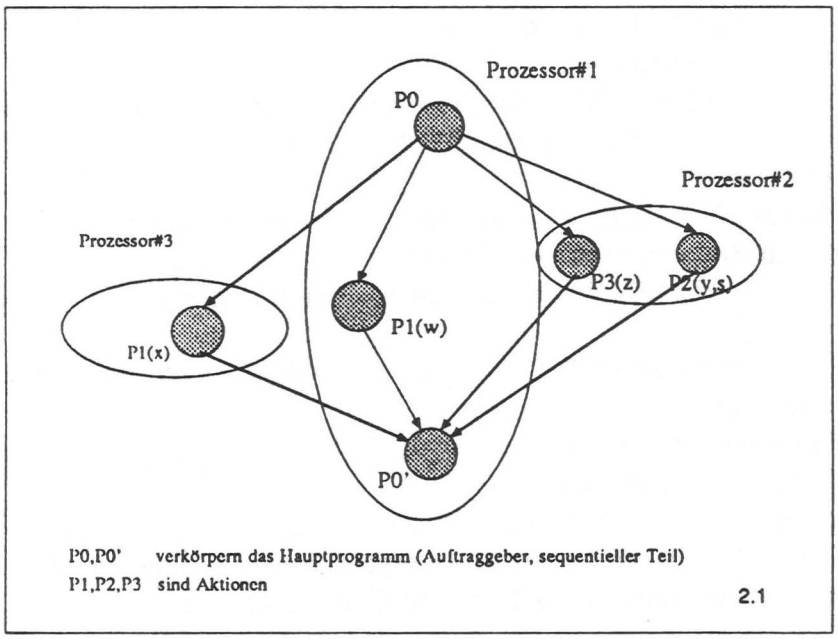

Abb. 2.1: Paralleler Ablauf

Zur Bedeutung: Auf jedem der Prozessoren (Prozessor#1,Prozessor#2,Prozessor#3) existiert ein E-Prozeß des E-Prozeß-Clusters. Für den Programmierer unterscheiden sich die E-Prozesse durch den Zustand der Daten. Nur beim Auftraggeber existiert eine

vollständige, aktuelle Version der Datenwerte. Im folgenden betrachten wir den Ablauf eines Programms aus der Sicht des Auftraggebers und eines Auftragnehmers (s.a. [Lu89a]).

Es sollen die Prozeduren p1,p2,p3 (s.a. Abb. 2.1) parallel ausgeführt werden. Dabei wird p1 mit zwei verschieden Parameterwerten aufgerufen.

Dann ergibt sich folgendes Bild:

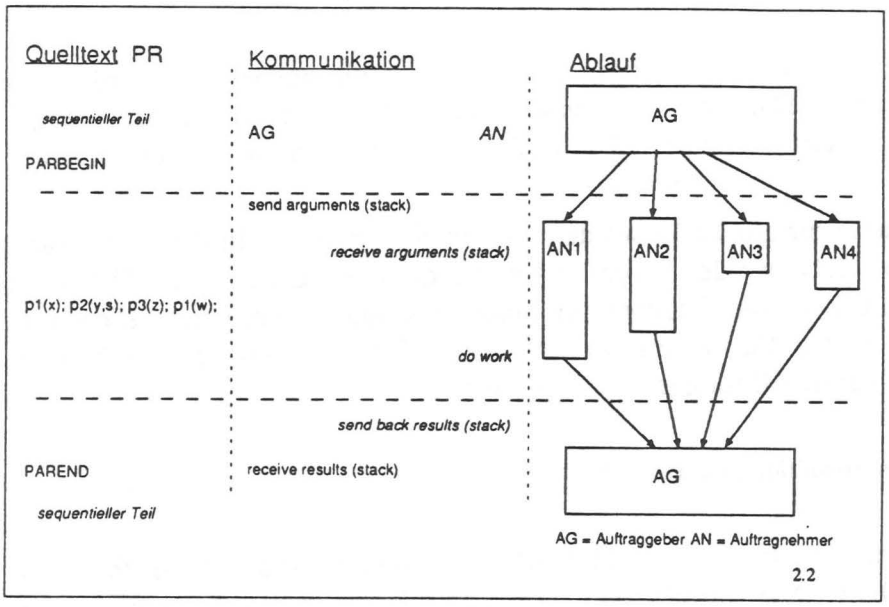

Abb. 2.2: Kontroll- und Datenfluß

Im Programm PR ist im Quelltext eine PARBEGIN ... PAREND-Klammer [Lu89a] vorhanden, welche angibt, daß die Prozeduren p1,p2,p3 parallel auf anderen Prozessoren ablaufen könnten. Im Bild sind maximal vier parallele Pfade möglich. Im Idealfall (bei fünf vorhandenen Prozessoren), würden also die Aktionen P1(x),P2(y,s),P3(z),P1(w) auf den Prozessoren (Auftragnehmer) AN(i), i=1..4, ablaufen, nachdem der Auftraggeber AG (der den vorangehenden sequentiellen Teil ausgeführt hat), die notwendigen Daten verteilt hat. Wenn eine Task nicht lokal ausgeführt wird, dann wird der Teil des Stacks als Nachricht an die Auftragnehmer versandt, der die Parameter der Task enthält. Der Auftraggeber sammelt bei PAREND die Ergebnisse der Auftragnehmer ein.

Die Aktionen des Auftraggebers für unser Beispiel sind, sofern genügend Prozessoren zur Verfügung stehen.

```
Versende Stack (x)    für p1 an E-Prozeß AN1;
Versende Stack (y,s)  für p2 an E-Prozeß AN2;
Versende Stack (z)    für p3 an E-Prozeß AN3;
Versende Stack (w)    für p1 an E-Prozeß AN4;

Warte auf Ergebnisse von {AN1,AN2,AN3,AN4};
Überprüfe Ergebnisbereiche;
Schreibe neue Werte;
```

Die Aktion eines Auftragnehmers:

```
Warte auf Botschaft;
führe Aktion aus;
sende Ergebnis zurück;
```

Beim Versenden müssen die Auftragsdaten beim Auftraggeber in Zuordnungstabellen festgehalten werden. Dazu gehören insbesondere Informationen darüber, welche Stackdaten an welchen Auftragnehmer (E-Prozeß) und damit auch an welchen Prozessor versandt wurden.

Die Parameter für die Tasks werden also zu Nachrichten. In die Parameterliste der Tasks (s.a.Abb. 2.4) werden vom Präprozessor die notwendigen Werte aus der zugehörigen Bereichs- und Taskbeschreibung eingefügt. Diese zusätzlichen Informationen bestimmen den Entstehungsort (Prozessor, E-Prozeß) einer Botschaft eindeutig. Dies ist wichtig für den Debugger und die Diagnose.

2.1. Expansion der Definition und des Aufrufs von Tasks

Im "Parallelisierungskonzept von ATTEMPTO-2" werden Prozeduraufrufe, die in einem PARBEGIN ... PAREND Block stehen, von einem Präprozessor expandiert. Neben den Aufrufen der Tasks müssen dabei auch ihre Definition berücksichtigt werden.

Anstatt auf Abb. 2.1 beziehen wir uns von nun an auf das daraus abgeleitete Schema eines Beispielprogramms, an dem die Expansionsidee gezeigt werden soll:

```
FROM somemodule IMPORT p2;    (*  Referenz zu einer Prozedur. Deklariert  *)
                              (*  als p2(x : INTEGER; VAR y : INTEGER);   *)

VAR
   x,y,z,w,s : INTEGER;

PROCEDURE p1(VAR x : INTEGER);       (*  Deklaration einer Modul-lokalen  *)
BEGIN                                (*  Prozedur                         *)
   ...
END p1;
```

```
BEGIN   (*  Hauptprogramm  *)
   ...
   PARBEGIN                        (*  Aufruf von p1 im parallelen Bereich  *)
      p1(y);
      p2(u,v);
      p3(z);
      p1(w);
   PAREND;
   ...
   p1(z);                          (*  Aufruf von p1 im sequentiellen Teil  *)
   ...
END Hauptprogramm.
```

Abb. 2.3: Paralleles Programm (vor Präprozessorlauf)

Der Präprozessor expandiert das Programm von Abb. 2.3 in den folgenden Text in Abb. 2.4. Auf die Einzelheiten werden wir im folgenden näher eingehen.

```
FROM PARLib     IMPORT ParBegin , ParEnd, ...;
FROM somemodule IMPORT p2;


PROCEDURE p1( VAR x : INTEGER );
BEGIN
   (*  Alte Prozedur p1 bleibt erhalten  *)
END p1;


PROCEDURE PARp1( FirstParam : DummyType;    (*  Parameterstackgrenze kennzeichnen   *)
                 Service    : PROC;         (*  Taskadresse                         *)
                 scopeInfo  : tScopeInfo;   (*  Info zu aktivem parallelen Bereich  *)
                 taskInfo   : tTaskInfo;    (*  Info zum zu aktivierenden Task      *)

                 x          : INTEGER;      (*  Expandierte Prozedurparameter:      *)
                 ADRx       : ADDRESS;      (*  Wert, Adresse und Grösse des ur-    *)
                 SIZEx      : CARDINAL;     (*  sprünglichen Parameters             *)

                 LastParam  : DummyType ); (*  Parameterstackgrenze kennzeichnen   *)


   (*  Neue Prozedur, die beim Auftraggeber für die Auftragsverteiluhg  *)
   (*  und beim Auftragnehmer für den Aufruf von p1 sorgt.              *)
   (*  FirstParam und LastParam dienen dem Verständnis und erleichtern  *)
   (*  die Eingrenzung des Bereichs der Parameterdaten auf dem Stack    *)

VAR
   StackUpperBound,
   StackLowerBound : ADDRESS;
```

```
BEGIN  (* PARp1 *)
   IF WhoAmI() = Auftraggeber THEN          (*  Funktioniert so in TDI    *)
      Service := PROC(PARp1);               (*  und Cambridge Modula-2.   *)
      StackUpperBound := ADR(FirstParam);   (*  Anfangs- und Endadresse   *)
      StackLowerBound := ADR(LastParam);    (*  der Parameter ermitteln   *)
      SendServiceRequest( UpperBound,LowerBound, ... );
   ELSE (* Auftragnehmer *)
      p1(x);                                (*  Service lokal erbringen   *)
      SendBackResults( ... );
   END;
END PARp1;


   (*  Wenn kein PARp2 importiert wird, dann muß hier PARp2 analog       *)
   (*  zur Expansion von p1 zu PARp1 erzeugt werden.                     *)


BEGIN (* Hauptprogramm *)
   ...                                (* Initialisierung des Prozeß-Systems  *)
   ParBegin( ... );
      PARp1( dummy,p1,scopeInfo,taskInfo,  y,ADR(y),SIZE(y),     dummy );
      PARp2( dummy,p2,scopeInfo,taskInfo,  u,v,ADR(v),SIZE(v),   dummy );
      ...
   ParEnd( ... );
   ...
   p1(z);                                          (* wird nicht expandiert *)
   ...
END Hauptprogramm.
```

Abb. 2.4: Paralleles Programm (nach Präprozessorlauf)

Für jeden Prozeduraufruf im parallelen Bereich muß also eine neue Prozedur definiert werden, welche die ursprüngliche Prozedur aufruft, wenn der E-Prozeß Auftragnehmer ist. Wenn der E-Prozeß Auftraggeber ist, muß sie die Botschaft versenden. Die Schlüsselworte PARBEGIN und PAREND werden durch die Laufzeitprozeduren ParBegin und ParEnd ersetzt.

Das Expansionskonzept des Präprozessors bietet folgende Vorteile:

(1) Importierte Prozeduren können in parallelen Bereichen verwendet werden.

(2) Die Expansion der Prozedurdefinitionen ist einfach, lesbar und verständlich (debugging).

(3) Die Portabilität wird erhöht, da der zu versendende Stackbereich in der Hochsprache Modula-2 bestimmt wird.

(4) Die Expansion der Prozeduraufrufe beschränkt sich auf PARBEGIN - PAREND Bereiche und beeinträchtigt damit nicht die normalen Aufrufe der Prozedur.

Im wesentlichen ergibt sich zur Laufzeit der folgende Nachteil:

Die Parameter einer Prozedur müssen zweimal auf den Stack gebracht werden.

2.2. Initialisierung des Prozeß-Systems

In einem Multirechnersystem werden Auftraggeber und Auftragnehmer auf verschiedene Prozessoren verteilt. Bevor die E-Prozesse ihre Aufgaben übernehmen können, müssen sie initialisiert werden. Die explizite Initialisierung eines E-Prozesses (s.a. Kap. 3 Die P-Laufzeitprozeduren) kann durch folgenden Programmtext beschrieben werden:

```
BEGIN  (* Hauptprogramm *)
   ...
   IF WhoAmI() # Auftraggeber THEN        (*  Auftragnehmer              *)
      InitializeAN;
      ReceiveMessages;                  (*  Auftragsannahme            *)
   ELSE
      InitializeAG;                     (*  Auftraggeber               *)
      DoSequentialWork;                 (*  Anwenderprogramm ...       *)
      ...
      ParBegin( ... );
         p1( ... );
         p2( ... );
         ...
      ParEnd( ... );
      ...
   END;                                 (*  ... Anwenderprogramm Ende  *)
END Hauptprogramm.
```

Abb. 2.5: Prozeßstart

Zuerst muß also der Status des E-Prozesses ermittelt werden, ob er Auftraggeber (AG) oder Auftragnehmer (AN) ist. Abhängig davon wird weiter initialisiert und das sequentielle Hauptprogramm bearbeitet, oder mittels *ReceiveMessages* (s.Abb. 3.1) auf Aufträge, d.h. auf *ServiceRequests* (vgl.a. Abb. B.2), gewartet.

2.3. Koordination und konsistente Datenhaltung

Die P-Laufzeitprozedur ParBegin markiert im expandierten Programmtext (s. Abb. 2.4), daß die nun folgenden Prozeduren parallel ausgeführt werden können. Ob der jeweilige E-Prozeß Auftraggeber (AG) oder Auftragnehmer (AN) ist, wird durch Abfrage einer zum E-Prozeß globalen Variablen ermittelt. Beschreibungen aller parallelen Bereiche werden in Tabellenform gespeichert. Die Tabelle errmöglicht es dem AG zu bestimmen, welche Aktion auf welchem Prozessor ausgeführt werden soll. Diese Tabelle wird vom Präprozessor (s. [Lu89a]) als Typ definiert und vom Konfigurator erstellt. Der Konfigurator ergänzt sie um die Zuordnungen von Aktionen zu virtuellen Prozessen. Beim Programmstart wird sie dann mit der aktuellen Identifikation des Prozessors (z.B. gethostname(2) [Uni79]) gefüllt. Diese Zuordnungen bleiben im Normalfall über die gesamte Bearbeitungszeit des Programms erhalten. Nur im Fehlerfalle und anschliessender dynamischer Rekonfiguration sollen sie verändert werden.

Vor der Abarbeitung des parallelen Bereichs erfragt ParBegin() im Auftraggeber die aktuell gültige Konfiguration und setzt damit die aktuellen Werte für die Aufrufparameter "Auftraggeber" und "Auftragnehmer" im *scopeInfo*. Der Auftraggeber

veranlaßt die Abarbeitung der Prozeduren eines parallelen Bereichs (Aktionen) auf anderen Prozessoren (ServiceRequest) durch Senden einer Botschaft mit den Parameterdaten des Aufrufs (Remote Service Invocation). Im Fall eines homogenen Mehrrechnersystems besteht diese Botschaft aus dem für die Task relevanten Stackinhalt des Aufrufs. Der Auftraggeber erwartet mittels ParEnd() die geänderten Werte der Variablen aller Auftragnehmer und bringt die Ergebnisvariablen bei sich konsistent auf den neuesten Stand. Nach dem Versenden des ServiceRequests bis zum Empfang der Ergebnisbotschaft erfolgt keine weitere Kommunikation mit den Auftragnehmern.

2.4. Bereichsglobale Daten

Damit die E-Prozesse keine inkonsistente Datenhaltung haben, sollte der direkte Zugriff eines E-Prozesses auf globale Daten unterbunden werden. Verschachtelte parallele Bereiche dürfen keine globalen Daten ändern, sondern nur die lokale Variablen der anstoßenden Task. Leider genügt es dafür nicht, die Tasks in eigenen Moduln zu deklarieren, die keine globalen Variablen importieren oder vereinbaren, da mit dem Import von Prozedurnamen nicht sichergestellt ist, daß die importierten Prozeduren keine globalen Variablen oder weitere Prozeduren mit globalen Variablen oder Zeigern benutzen. Also muß die Benutzung von globalen Daten bei Auftragnehmern durch andere Mechanismen erkannt und entsprechend behandelt werden.

(1) Zur Laufzeit könnte dies durch eine Memory Management Unit (*MMU*) analog zum *page fault* erkannt werden. Dies ist aber nicht allgemeingültig, da die Binde- und Ladekonventionen sowie der Zugriffsschutz des Zielsystems eine bedeutende Rolle spielen.

(2) Zur Übersetzungszeit könnte dies ein aufwendiger Präprozessor übernehmen (s.a. Kapitel 4).

(3) Der Programmierer kennzeichnet einen Zugriff auf globale Daten explizit als unbedenklich (s.a. Abschnitt 4.2).

2.5. Vergabe der Aufträge zur parallelen Ausführung

Kommt nun der Auftraggeber AG im Verlauf der sequentiellen Bearbeitung an ein PARBEGIN, bzw. einen Aufruf von ParBegin(), dann müssen anschliessend die Stackdaten der aufgerufenen Prozeduren versandt werden, ohne daß die Prozeduren (Tasks) lokal bearbeitet werden. Kommt der Auftraggeber AG dann zum PAREND, bzw. zum Aufruf von ParEnd(), so muß er auf die Ergebnisse der einzelnen Aktionen warten. Die Laufzeitroutinen ParBegin, ParEnd müssen und dürfen nur vom Auftraggeber abgearbeitet werden.

Die Botschaften zwischen Tasks bestehen aus den Aufrufparameter der Prozeduren im parallelen Bereich. Damit diese Botschaften eindeutig zu identifizieren sind (Konsistenzprüfung, Debugging) müssen sie zusätzlich zu den eigentlichen Prozedurparametern sowohl Absender und Empfänger, als auch Ort (im Kontrollfluß) und

Zeitpunkt (dynamische Verschachtelung) des Entstehens der Botschaft enthalten (s.a. Abschnitt 3.2). Deshalb müssen diese Daten vom Präprozessor als zusätzliche Parameter textuell vorgesehen und im späteren Ablauf entsprechend eingesetzt werden.

Zur konsistenten Datenhaltung beim Auftraggeber müssen die Ergebnisse der Aktionen übernommen werden. Der Auftragnehmer muß die Werte aus den Ergebnisbotschaften in die lokalen Variablen kopieren. Dazu benötigt er zusätzlich die Größe und Adresse der Datenvariablen. Deshalb muß der Präprozessor beim Prozeduraufruf einen VAR- Parameter durch dessen Adresse (VAR), Größe (TSIZE(Parameter)) und den Wert selbst ersetzen.

Wir wollen nun die Erzeugung des Aufrufstacks genauer betrachten. Um die Parameter für eine Prozedur von Modula-2 aus auf den Stack zu kopieren und danach den Service aufrufen zu können, müssen vom Compiler folgende Forderungen erfüllt werden:

(1) Alle Parameter einer Prozedur müssen auf dem Stack übergeben werden.

(2) Der vom Compiler erzeugte Code muß die Konsistenz von Stack und Registern nach der Rückkehr aus einer Prozedur garantieren, unabhängig davon, ob die erwartete Anzahl der Parameter der tatsächlich übergebenen Anzahl entspricht (wegen des Aufrufs einer Prozedur ohne Parameter, obwohl in der Definition welche vorgesehen sind s.Abb. 2.4, service:=PROC(...);).
Diese Voraussetzung dürfte wohl von den meisten Modula-2 Compilern erfüllt sein (TDI Modula-2 [TDI] und Cambridge Modula-2 [Cam] erfüllen diese Forderung, obwohl beide eine völlig unterschiedliche Vorgehensweise bei der Verwaltung des Linkregisters haben).

(3) Auf dem Stack müssen (in Richtung wachsender Stack) zuerst die Parameter und dann die lokalen Variablen einer Prozedur liegen. Dies garantiert, daß sich die Stackadressen der Parameter unabhängig von der Anzahl und Größe der lokalen Variablen berechnen lassen. (Erfüllt in TDI-Modula und Cambridge Modula-2, siehe Testprogramm im Anhang Abb. A.1)

(4) Lokale Variable werden in der Reihenfolge ihrer Definition (oder genau umgekehrt) auf dem Stack angelegt und nicht in Registern gehalten. Über den ADR Operator läßt sich damit der Bereich der Parameter auf dem Stack bestimmen (erfüllt in TDI und Cambridge Modula-2).

Ein Testprogramm, das überprüft ob diese Forderungen erfüllt sind, ist in Anhang A aufgeführt.

2.6. Parallele Ausführung der Aufträge

Die Auftragnehmer warten in *ReceiveMessages* (s.Abb. 3.1) auf eine Botschaft. Der empfangene Stack wird dann auf Konsistenz überprüft (s.u.). Danach wird der gewünschte Service, der über die Serviceadresse (s.a. 2.4) bekannt ist, mittels *CallService* erbracht.

In den Info-Parametern (*scopeInfo,taskInfo*) steht, wer die Botschaft versandt hat. *MyIdForThisTask* sei die Kennung für den Auftragnehmer dieser Teilaufgabe, wie sie von Konfigurator und Auftragsverteilung festgelegt wurde. Diese redundante Information wird vom Empfänger zur Konsistenzprüfung verwendet. Sind z.B. *MyIdForThisTask* und der in den Info-Parametern vorgesehene Auftragnehmer unterschiedlich, wird eine Diagnose angestoßen.

Die Task p1 (s.a. Abb. 2.3) wird sequentiell durchlaufen. In der zu PARp1 expandierten Task wird im Auftraggeber die Serviceadresse festgestellt und dann der ServiceRequest versandt. Beim Auftragnehmer dagegen wird in PARp1 gesprungen und dann die ursprüngliche Prozedur p1 ausgeführt und abschließend das Ergebnis zurückgeschickt.

Im folgenden wird skizziert wie der Stackmechanismus zum Anstoßen der geforderten Task beim Auftragnehmer benutzt wird. Die Parameterdaten des Aufrufstacks wurden vorher in einer Botschaft empfangen (vgl. Abb. 2.3, 3.1).

```
TYPE
   tMessage = POINTER TO ARRAY[0..MAXLEN] OF WORD;

PROCEDURE CallService(mess : tMessage);
CONST
   Offset = ... ;  (*  Z.B. 0 für TDI, 24 für Cambridge Modula-2          *)
   MAX    = ... ;  (*  MaxMessLength für TDI, 0 für Cambrigde             *)
   Infix  = ... ;  (*  TRUE,  falls Return-Adressen zwischen Parametern  *)
                   (*         und lokalen Variablen einer Prozedur auf    *)
                   (*         dem Stack liegen.                           *)
                   (*  FALSE, falls Return-Adressen vor Parametern und    *)
                   (*         Parametern auf dem Stack liegen (Cambr.)    *)


TYPE
   StackSpace = ARRAY[0..MAX] OF WORD;

VAR
   Service      : PROC;
   adr          : ADDRESS;
   indx         : CARDINAL;
   LastLocalVar : StackSpace;
```

```
BEGIN   (*  CallService  *)

   Service := ServiceRoutineAddr(mess);    (*  Die Startadresse des ge-   *)
                                           (*  wünschten Service muß in   *)
                                           (*  der Nachricht kodiert sein *)
   IF Infix THEN
      adr := ADR(LastLocalVar);
   ELSE
      adr := ADR(LastLocalVar) - Offset - LengthInWords(mess);
   END;

   FOR indx:=1 TO LengthInWords(mess) DO   (*  Die Nachricht wird auf den *)
      adr^[indx] := mess^[indx];           (*  Stack kopiert              *)
   END;


   (*  Nachdem die Nachricht auf den Stack gebracht wurde, kann der      *)
   (*  Service aufgerufen werden. Der Service ist die vom Präprozessor   *)
   (*  expandierte Prozedur (PAR...), die die eigentliche Prozedur       *)
   (*  ausführt und die Ergebnisse (den Stack) zurücksendet              *)

   Service;

END CallService;
```

Abb. 2.6: Aktion anstoßen

Auf dem Stack liegen nach unserer Konvention sowohl die Adressen und Längen (SIZE) als auch die Werte der Parameter. Die Werte werden mit einer stackrelativen Adressierung gelesen. Natürlich können sie auch verändert werden. Bei den meisten Sprachen werden beim Rücksprung diese neuen (lokalen) Werte durch die Korrektur des Stackpointers vergessen. In unserem Fall werden sie am Prozedurende an den Auftraggeber im *Ergebnis-Stack* zurückgesandt. Der Ergebnis-Stack entspricht in der Struktur dem Aufruf-Stack. Somit kann in ParEnd() für das Aktualisieren von Variablen einfach auf deren Adressen und Typgröße zugegriffen werden. Neben den *reinen* Daten sollte die Identifikation der Auftraggeber und Auftragnehmer immer mit versandt werden (robuste Datenstrukturen).

3. Die P- Laufzeitprozeduren

Die P-Laufzeitroutinen sollen in einem Programmmodul (*PARLib*) zusammengefaßt werden. Der Präprozessor erzeugt die notwendige Anweisung zum Import, so daß die erforderlichen P-Laufzeitroutinen im expandierten Quelltext benutzt werden können und die Initialisierung dieses Moduls zuerst erfolgt. Die im Abschnitt 2.2 beschriebene Initialisierung des Prozeßsystems wird in eine Prozedur *ParInit* dieses Moduls umgewandelt und dann als erste Prozedur im Block und damit als erste des gesamten Programms aufgerufen.

3.1. Hilfsroutinen zur Kommunikation und Synchronisation

Da alle Prozesse denselben Code haben (E-Prozesse), wird im Programm zwischen Auftragnehmer und Auftraggeber unterschieden. Beim Auftraggeber müssen zuerst alle Stackdaten des parallelen Bereichs als Botschaft (message) verschickt werden, bevor er selbst einen Auftrag übernehmen darf. Dafür muß der Konfigurator sorgen. Denn sonst würden die Aktionen sequentiell auf dem Auftraggeber bearbeitet.

Im sequentiellen Hauptprogrammteil der E-Prozesse führen die Auftragnehmer die Prozedur *ReceiveMessages* (s.Abb. 2.5) aus.

```
PROCEDURE ReceiveMessages;
BEGIN
   LOOP
      WaitForMessage(mess);              (*  Warten auf Aufträge          *)
      IF WhoAmI() = Auftraggeber THEN    (*  Nur Auftragnehmer können     *)
         Error(...);                     (*  Aufträge annehmen            *)
      END;
      CallService(mess);                 (*  Auftrag ausführen            *)
   END;
END ReceiveMessages;
```

Abb. 3.1: Prozedur ReceiveMessages

Am Beginn des parallelen Bereichs (s.a.Abb. 2.4) setzt der Auftraggeber mit dem Aufruf von ParBegin() die für den folgenden parallelen Bereich geltenden aktuellen Parameter (Modulnamen, Bereichsnummer und Auftraggeberidentifikation) fest.

Anmerkung: An diesem Punkt im Befehlsfluß ist es sinnvoll aus Fehlertoleranzgründen einen Watchdog-Timer (*Timeout*) für den parallelen Bereich zu starten.

Im folgenden (s.a. Abb. 3.2) wird die vom Konfigurator und der Auftragsverteilung erstellte Aktionsablaufplanung für die aktuell gültige Prozessorkonfiguration, mit *DispTable* bezeichnet. In ParBegin() wird mittels *SearchInDispTable* in der DispTable nach der aktuell gültigen Liste der virtuellen Prozesse gesucht. Nachdem die für diesen parallelen Bereich benutzten virtuellen Prozesse bestimmt sind, wird mit *WhereLoaded* erfragt, welche Prozessoren der Systemlader für diesen E-Prozeß-Cluster ausgewählt hat. Damit sind die Identifikationen der Auftragnehmer *ANs[]* für diese Aufgabe und damit die Liste der Zielprozessoren *PList* bestimmt. Diese werden dann beim Versenden der Botschaft als Ziel eingesetzt und mit der anderen Information in den Zuordnungstabellen

für den *Check1* von *ParEnd* festgehalten.
Am Ende des parallelen Bereichs sammelt im Auftragnehmer die P-Laufzeitprozedur ParEnd die Ergebnisse ein:

```
VAR
   DispTable : ARRAY [1..NumberOfParRegions] OF
                  ARRAY [1..MaxThreads] OF tTaskDescription;

   (*  Dispatching-Tabelle für alle parallelen Bereiche  *)
   (*  der momentan gültigen Prozessor-Konfiguration     *)

   AN : ARRAY [1..MaxThreads] OF ProcessorName;


   ...

PROCEDURE ParBegin(ModuleName : t1; ParRegionNumber : CARDINAL);
VAR
   indx  : CARDINAL;
   PList : POINTER TO ARRAY [1..MaxThreads] OF tTaskDescription;
BEGIN

   (*  ParBegin-ParEnd Block in DispTable suchen                *)

   PList := SearchPListInDispTable(ModuleName,ParRegionNumber);

   (*  Ermittle für jeden E-Prozeß den zugeordneten Prozessor   *)

   FOR indx:=1 TO MaxThreads DO
      AN[indx] := WhereLoaded(PList^[indx]);
   END;
   ...
END ParBegin;


PROCEDURE ParEnd(ModuleName : t1; ParRegionNumber,
                                  ActualNumberOfThreads : CARDINAL);
VAR
   Thread : CARDINAL;
BEGIN
   FOR Thread:=1 TO ActualNumberOfThreads DO
      GetResultMessage;
      Check1;           (*  Botschaft: AG, AN, Bereich korrekt?  *)
      Check2;           (*  Ergebnis : Race condition?           *)
      UpdateResults;
   END;
END ParEnd;
```

Abb. 3.2: Prozeduren ParBegin und ParEnd

Die Prozedur *Check1* bestimmt, ob die Botschaft *mess* die erwarteten Werte bezüglich Sender, Empfänger, paralleler Bereich und Task hat, oder ob ein Timout aufgetreten ist.

In *Check2* wird überprüft, ob in diesem ParEnd derselben Adresse mehrfach ein Wert zugewiesen werden soll, was einen Fehler bei der Programmierung vermuten läßt.

Alle für die Parallelisierung benötigten Daten müssen auch für die Diagnose und Rekonfiguration gesammelt werden, wenn nicht nur Jobs, sondern auch die Tasks fehlertolerant ablaufen sollen. In ATTEMPTO-2 sollen die Daten für beide Zwecke genutzt werden.

Anmerkung: In ATTEMPTO-2 benutzen Auftragnehmer und Auftraggeber die gepufferte und über Timeout gesicherte Kommunikation von ATTEMPTO [Dal87] . Schätzwerte für Timeouts können aus der Abarbeitung des sequentiellen Programmlaufs für den Konfigurator gewonnen werden. Sie werden in der Ablaufplanliste für die entsprechende Laufzeitroutine aufbewahrt.

3.2. Laufzeitroutinen für die dynamische Parallelisierung

Mit der bisher betrachteten Parallelisierung ist bei vielen Problemstellungen nicht mit einer maximalen Auslastung des Systems zu rechnen. Die Auslastung in einem größeren Netz könnte durch eine dynamische Auftragsverteilung verbessert werden.

Die wesentliche Erweiterung besteht darin, daß in den Prozeduren, die in den parallelen Bereichen aufgerufen werden, wiederum parallele Bereiche eingeschachtelt werden können. Die parallelen Bereiche entwickeln sich nun dynamisch und müssen am jeweiligen ParEnd()- Aufruf beendet werden. Die dynamische Form verlangt, daß in den E-Prozessen bei ParEnd() nicht nur auf die Ergebnisse von Aktionen gewartet wird, sondern auch neue Anforderungen von Aktionen erkannt und umgesetzt werden können.

Wesentliche Voraussetzung im dynamischen Fall ist das Vermeiden von Deadlocks. Deadlockfreiheit ist aber nur dann gewährleistet, wenn kein Prozeß, der eine Nachricht senden will, blockiert wird. Auch soll im dynamischen Fall ein Programm reproduzierbar (entweder korrekt oder fehlerhaft) ablaufen.

Um eine dynamische Schachtelung von PARBEGIN und PAREND zu ermöglichen, muß ein Prozessor, auf dem ein E-Prozess abläuft, der gerade im ParEnd auf Ergebnisse wartet, die Möglichkeit haben, Aufträge anderer Prozessoren anzunehmen und auszuführen. Die Ergebnisse können in zeitlich nicht vorgegebener Reihenfolge zurückkommen und müssen deshalb in dieser willkürlichen Reihenfolge verarbeitet werden. Dies ließe sich realisieren, indem man auf Dienstleistungen des Betriebssystem (wie z.B. fork(2) [Uni79]) zurückgreift. Zur Bearbeitung mehrerer ServiceRequests können auch Modula-2 Leichtgewichts-Prozesse (Coroutinen) eingesetzt werden. Diese Lösung ist allgemeiner, da die dynamische Verschachtelung im Programm mit der bekannten Information selbst begrenzbar wird. Zudem wird der Overhead geringer als bei einer Prozeßgenerierung durch das Betriebssystem.

In Anhang B wird eine betriebssystemunabhängige Lösung in Modula-2 vorgestellt, die sowohl effizienter als auch portabler sein dürfte. Der Wunsch ist es, dabei völlig ohne in Assembler geschriebene Teile auszukommen und höchstens systemabhängige Konstanten (ByteOffsets usw.) zu verwenden.

Ob dies erreicht werden kann hängt allerdings davon ab, in wie weit der verwendete Modula-2 Compiler den in Abschnitt 2.5 spezifizierten Restriktionen genügt. Diese Restriktionen werden jedoch von den uns bekannten Compilern erfüllt (s.a.Abb. 2.6).

Mit den Skizzen und Spezifikationen im Anhang B.2 soll gezeigt werden, daß auch für den dynamischen Fall die Implementierung einfach machbar ist, und wie sie durchgeführt werden kann.

4. Parallel ausführbare Unterprogramme

Die Unterprogramme der Laufzeitbibliothek werden als vorübersetzte Objekte in Bibliotheken abgelegt. Identifiziert werden sie durch ihre Namen. In sequentiellen Programmen werden alle referenzierten Objekte kopiert und zu einem ablauffähigen Programm zusammengebunden. Die Aufrufe ein und desselben Objekts werden an *dieselbe* Adresse (Daten, Programmcode) gerichtet (gelinkt).

Wir betrachten die Gesamtheit der *P-Laufzeitunterprogramme* in der *parallelen* Bibliothek als für den Benutzer transparente Zwischenschicht, welche der Parallelisierung und Fehlertoleranz dient. Dienste aus der sequentiellen Bibliothek sind dazu zum Teil geeignet zu erweitern.

Viele Bibliotheksmodule sind als abstrakte Datentypen (*ADT*) implementiert (s.a. [DLR88]). Die Dienste werden durch den Aufruf von Prozeduren angefordert und ändern meist den Zustand des ADT. Bei Prozeduren, die zu Tasks werden können, müssen diese Zustandsvariablen in die Parameterlisten der Prozeduren übernommen werden. Dies bedeutet, daß die Quellen dieser Bibliotheksmodule überarbeitet werden müssen. Wenn bestimmte Bibliotheksroutinen nicht parallel ausgeführt werden dürfen, also nicht in Tasks verwendbar sind, dann muß der Programmierer eine neue parallelisierbare Schnittstelle schaffen.

Eine Idee, die Integration bestehender Bibliotheksmodule in das Konzept zu vereinfachen, besteht darin, Zusicherungen über die problemlose Verwendung von ausgewählten Moduln (z.B. Storage) abzugeben. Diese sogenannte pe-Eigenschaft (s.u.) wird korrespondierend zu jedem Modul in einer Datei *.pe* gespeichert, analog zu den Typdefinitionen von Definitionsmoduln in *.sym*-Dateien. Die Benutzung von globalen Variablen in einer Task kann dann geprüft (s.a. Abschnitt 4.2) und gegebenenfalls zugelassen werden. Alle Moduln wie *Storage*, die *Resourcen* verwalten, müssen von allen E-Prozessen initialisiert werden. Die Verwendung von Zeigern soll bei der Übersetzung immer zu *Warnungen* führen, auf expliziten Wunsch aber zugelassen werden.

Anmerkung: Bei Zugriffen auf die zum E-Prozeß-Cluster globalen Dateien aus verschiedenen Tasks wird der gegenseitige Ausschluß durch das ATTEMPTO-Protokoll [Dal87] erreicht.

4.1. Zusicherung der parallelen Ausführbarkeit von Unterprogrammen

Eine Prozedur ist parallel ausführbar (*pe, parallel executable*), wenn sie nur auf lokale Variable und Parameter zugreift. Sie darf also nicht direkt oder indirekt (über von ihr aufgerufene Prozeduren) auf globale Variable oder Bereiche des Heaps lesend oder schreibend zugreifen.

Im folgenden sollen zwei Verfahren vorgestellt werden, die Modula-2 Prozeduren anhand des Quelltextes automatisch in *pe* und *nicht-pe* klassifizieren. Beide sind voll aufwärts und abwärts kompatibel zum bestehenden Modula-Compiler (*m2c*) und geben die Idee der separaten Kompilation von Moduln nicht auf. Ausgangspunkt ist bei beiden Methoden die Erweiterung der Syntax eines Definitionsmoduls durch einen Pseudokommentar, der von einem Scanner (*DefChecker* genannt) erkannt wird. Dabei werden bei der einfachen Überprüfung alle exportierten Prozeduren des Moduls als *pe* deklariert. Im zweiten Fall werden einzelne als *pe* deklariert. Bei der Kompilation der zugehörigen Implementationsteile muß dann der *ImpChecker* (ein Scanner im einfachen Fall, ein Parser im komfortablen Fall) sicherstellen, daß die Implementation einer als *pe* deklarierten Prozedur auch wirklich *pe* ist.

Die erste Idee benutzt einen einfachen, leicht zu erstellenden Scanner. Sie bringt aber nur wenig Komfort: Prozeduren bestehender Bibliotheken können nur schwer als *pe* erklärt und überprüft werden.

Die zweite Methode ist eine Weiterführung der ersten Idee. Sie benötigt jedoch statt eines Scanners einen Modula-2 Parser zur Konsistenzprüfung. Auch bestehende Bibliotheksprozeduren lassen sich, wenn sie *pe* sind, nachträglich als *pe* deklarieren und überprüfen.

4.2. Einfache Überprüfung der Zusicherung

Ein Pseudokommentar am Beginn eines DefinitionModuls (*.def* Modul) deklariert alle Prozeduren des Moduls als *pe*.

```
DEFINITION MODULE  mymodule (*$pe*) ; . . .
```

Nach dem Aufruf des Modula-Compilers *m2c* wird automatisch der DefChecker aufgerufen, der ein File *.pe* erzeugt, wenn der Pseudokommentar gefunden wird und das Schlüsselwort VAR nicht im *.def*-File auftaucht. (Siehe Abb. 4.1)

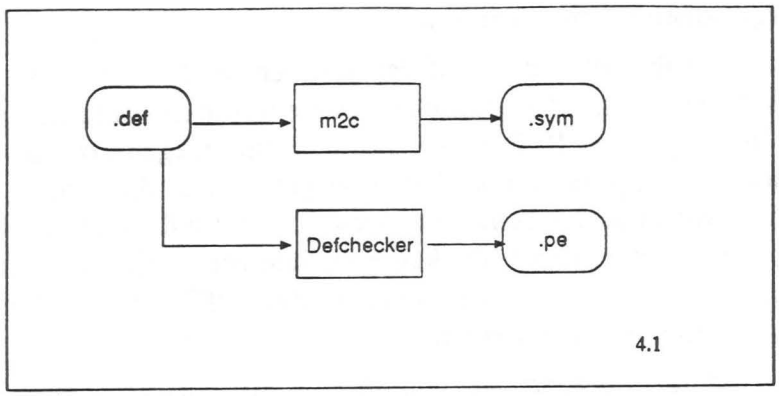

Abb. 4.1: Erstellung einer pe Zusicherungsdatei

Der *.pe*-File enthält nur den Modulschlüssel des zugehörigen .def-Files. Bei der Kompilation des zugehörigen Implementation Moduls (*.mod*) muß die pe-Eigenschaft aller exportierten Prozeduren natürlich sichergestellt werden.

Dazu wird nach m2c, wenn ein .pe-File existiert, automatisch der ImpChecker für das .mod-File aufgerufen, der das von m2c erzeugte .lnk-File sofort wieder löscht, falls die pe-Eigenschaft nicht sichergestellt werden kann (s. Abb. 4.2).

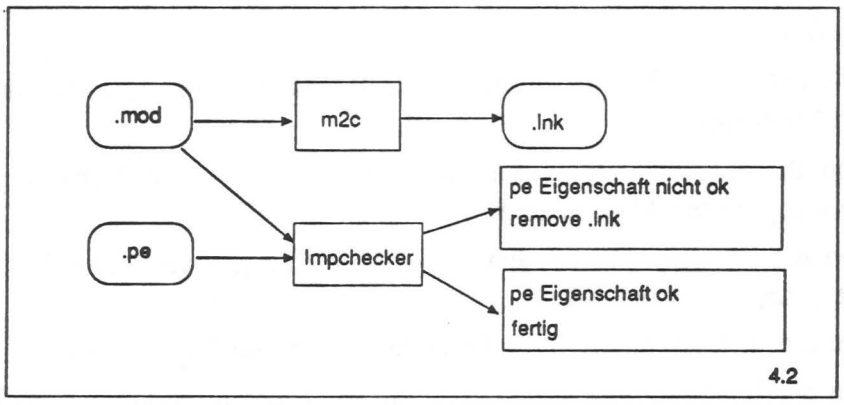

Abb. 4.2: Überprüfung der pe Zusicherung

Ist im Implementationsteil kein Pseudokommentar enthalten, der die *pe*-Eigenschaft zusichert, so überprüft der ImpChecker das .mod-File nach den folgenden drei einfachen Kriterien:

(1) Eine Zeigerdereferenzierung ('^') darf im Modultext (außer in Kommentaren und Strings) nicht auftauchen. Dadurch wird sichergestellt, daß über übergebene Adressen kein Zugriff auf den Heap, globale Variable oder lokale Variable dynamisch übergeordneter Prozeduren erfolgt.

(2) Das Schlüsselwort VAR darf niemals auf der obersten Modul-Ebene auftauchen. (Der ImpChecker muß also die statischen Prozedurschachtelungen erkennen können, was kein großes Problem darstellt. Im Gegensatz zu einem Präprozessor kann der ImpChecker ja davon ausgehen, daß die Syntax korrekt ist, er also keine syntaktischen Fehler berücksichtigen muß. Der ImpChecker wird ja erst nach m2c aufgerufen.) Da keine moduleigenen, globalen Variablen definiert werden (für das *.def*-File hat dies bereits der DefChecker sichergestellt) können solche auch nicht benutzt werden.

(3) Es dürfen nur Module importiert werden, für die ein entsprechendes *.pe*-File existiert. Somit ist sichergestellt, daß auch auf globale Variable anderer Module direkt oder indirekt nicht zugegriffen wird.

4.3. Komfortable Überprüfung der Zusicherung

Statt alle Prozeduren eines Moduls als pe zu deklarieren, kann man auch einzelne Prozeduren als pe deklarieren:

```
PROCEDURE (*$pe*) someprocedure ( ... ) ;
```

Statt nur ein *.pe*-File mit dem Modulschlüssel anzulegen, muß der DefChecker nun im *.pe*-File auch noch die Namen aller als pe deklarierten Prozeduren ablegen.

Der ImpChecker ist nun allerdings wesentlich schwieriger zu realisieren. Er entscheidet nach folgendem Verfahren, ob ein Implementation Modul die *pe-* Zusicherungen im Definition Modul erfüllt:

(1) Erzeuge eine Liste aller Prozeduren des Moduls (aller statischen Level) und aller importierten Prozeduren.

(2) Streiche von dieser Liste alle importierten Prozeduren, deren Name nicht im .pe-File des importierten Moduls steht.

(3) Streiche von dieser Liste alle lokalen Prozeduren, die globale Variable (importierte oder eigene) direkt benutzen.

(4) Streiche von dieser Liste alle lokalen Prozeduren, die den Zeiger benutzen.

(5) Streiche von der Liste alle lokalen Prozeduren, die Prozedurvariable aufrufen.

(6) REPEAT
Streiche von der Liste alle Prozeduren, die Prozeduren aufrufen,
die nicht in der Liste stehen.
UNTIL Liste bleibt stabil;

(7) Sind noch alle im .pe-File stehenden Prozeduren in der Liste.
Ja: Fertig.
Nein: Lösche das .lnk-File und gebe Fehlermeldung aus.

Insbesondere der dritte Schritt dürfte ohne einen Modula-2 Parser schwierig sein (man denke nur an WITH Anweisungen).

Anmerkung: Aus pragmatischen Gründen (u.a. Verwendung von Zeigern) werden die Forderungen in den Punkten (3)-(5) gelockert. Bei einer expliziten Zusicherung durch den Programmierer mittels eines Pseudokommentars für jede Variable, wird vom Def- und ImpChecker an diesen Stellen nur eine Warnung ausgegeben.

5. Konfigurierung und Verteilung eines parallelen Programms

Die Information, welche die parallele Struktur eines verteilten Programms beschreibt, wird im folgenden mit *Parallelstrukturinformation* bezeichnet. Sie enthält den Aufrufgraphen, den Umfang der Kommunikation und beschreibt in Form einer Tabelle die parallelen Einheiten (Tasks) und welche Beziehung zwischen ihnen besteht. Die parallelen Bereiche werden in der Reihenfolge ihres Auftretens im Programmtext in diese Tabelle eingetragen. Diese Tabelle wird von uns *Bereichskonfiguration* (*configuration data*) genannt. Jeder Eintrag für einen Bereich wird als *Scope-Information* bezeichnet.

Die aktuelle Verteilung zur Laufzeit erhalten wir erst nach folgenden Arbeitsschritten:

(1) Extrahieren der Parallelstrukturinformation (*.pre*) durch einen Präprozessor

(2) Erzeugen von Profil-Information (*.prf*) durch einen Profilierungslauf [Gra82,Uni79] des sequentiellen Programms.

(3) Erstellen des Aktionsablaufplans durch den Konfigurator.

(4) Zuordnen der Tasks zu Prozessen in der Taskkonfiguration (*.cfd*)

(5) Abbilden der Prozesse auf die Prozessoren des Prozessornetz durch den Parallel Dispatcher (Mapper) des Systems.

(6) Verteilen auf die Prozessoren durch den Systemlader.

(7) Rekonfigurieren durch Ändern der Tabellen (Aktionsablaufplan) zur Laufzeit.

(8) Auslesen der Tabelle im aktiven E-Prozeß.

Bis zur Ausführung von verteilten Programmen sind also zumindest die in Abb. 5.1 aufgeführten Werkzeuge beteiligt.

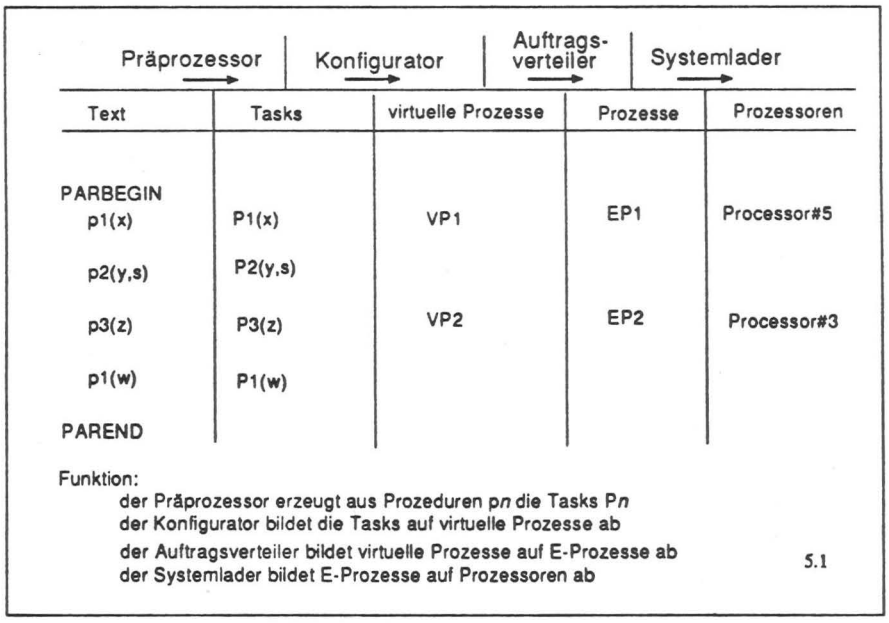

Abb. 5.1: Werkzeuge

5.1. Konfigurierung

Die Aufgabe des Konfigurators ist es, aus der Struktur eines verteilten Programms einen Aktionsablaufplan (Konfiguration) abzuleiten. Dabei spielen die Einheiten des Programms, deren Ablaufumgebung und Verbindungen eine Rolle. Die Zuordnung (*binding*) der einzelnen Tasks an die Prozessoren erfolgt in zwei Phasen.

(1) in der Vorbereitungsphase wird eine Konfiguration erstellt

(2) in der Laufzeitphase wird die Konfiguration auf die vorhandenen Prozessoren abgebildet.

Die Bindung der Tasks an die E-Prozesse wird vom Konfigurator vorbereitet, vom Auftragsverteiler bestimmt und im Aktionsablaufplan - der erweiterten Konfigurationstabelle - festgehalten. Der voraussichtliche Nutzen durch Verteilen eines Programms muß dabei vom Konfigurator ermittelt werden. Die maximale Dauer eines Programms von einem Kontrollflußpunkt k(0) am Anfang eines parallelen Bereichs bis zum Kontrollflußpunkt k(1) am Ende des parallelen Bereichs sollte bei einer Verteilung offensichtlich geringer sein, als wenn die Tasks sequentiell lokal durchlaufen würden. Die Zeit im verteilten Falle ergibt sich als die Summe der Zeiten für das Versenden t(send) der Daten nach k(0), warten auf das Ergebnis der längsten Taskausführung t(remote work) und Empfangen t(receive) der Ergebnisse bei k(1). Sie sollte offensichtlich kleiner sein als die lokale Bearbeitungszeit t(local work) des Bereichs.

```
t(send) + t(receive) + t(remote work) < t(local work)
```

In der ersten Phase werden die Prozedurlaufzeiten durch eine *sequentielle* Ausführung des Programms mit *Laufzeitprofil* (profiling) ermittelt. Mit der beim Profiling ebenfalls aufgezeichneten Aufrufhäufigkeit kann über die Anzahl und Größe der Parameter beim Prozeduraufruf der Datenfluß zwischen potentiellen Auftraggebern und Auftragnehmern geschätzt werden. Die aufsummierten Werte der Laufzeiten der Prozeduren in den Tasks werden dann als Schätzung der zu erwartenden Tasklaufzeit benutzt. Diese Daten gehen als zusätzliche Knoten- und Kantengewichte in den zu partitionierenden Strukturgraphen ein. Daraus erstellt der Konfigurator einen Aktionsablaufplan für die Tasks des verteilten Programms (s.a. [Lu89b]).

Der Typ der Konfigurationstabelle wird vom Präprozessor (s.a. Abschnitt 2) definiert. Der Konfigurator füllt die Tabelle mit den Daten aus der Parallelstruktur-Informationsdatei (*.pre*) über die parallelen Bereiche und den darin enthaltenen Tasks und ergänzt sie um die Profil-Information (*.prf*) aus dem Profilierungslauf.

Anmerkung: Unterstützt das vorhandene Betriebssystem oder die verwendeten Übersetzungswerkzeuge kein Profiling, dann kann dies durch den Präprozessor durch zusätzlich eingefügten Code beim Aufruf und Verlassen von Prozeduren erreicht werden. Die Frontends (Compiler) des von uns verwendeten Amsterdamer Compiler Kits (ACK) [Tan83] z.B. erzeugen die Prozeduraufrufe *procentry* und *procexit*. Normalerweise enthalten diese Unterprogramme nur einen Rücksprung. Sie sind fürs Profiling (s.a. [Lu89b]) bestens geeignet.

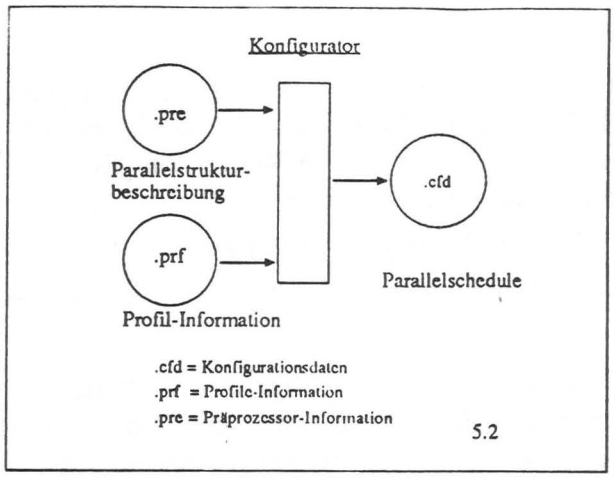

Abb. 5.2: Programmkonfiguration

Aus der Information der Dateien zur Parallelstruktur und dem Ablaufprofil mit den Dateinamenserweiterungen *.pre*, *.prf*, wird vom Konfigurator ein statisches Schedule - eine Taskkonfiguration - erstellt, und in der Konfigurationsdatei (*.cfd*) ausgegeben.

Der vom Konfigurator erstellte Aktionsablaufplan (*.cfd*) beruht auf einer vom Präprozessor definierten Tabelle, die als externes Objekt definiert wird und damit beim Linken zum Programm gebunden wird. Der Binder legt dann das Objekt an eine feste programmrelative Adresse. Die Adresse dieses eindeutig vereinbarten Namens kann als globale Variable in den unterstützenden P-Laufzeitroutinen angesprochen werden. Deshalb ist es möglich, die im *.cfd*-File vom Konfigurator abgelegten Daten beim Programmstart einzulesen. Der Konfigurator ordnet die Tasks statisch einer maximalen Anzahl *Nmax* von virtuellen Prozessen (s. Abschnitt 1.2) zu, die sich ableiten läßt aus der maximalen Breite aller parallelen Bereiche (s.a. [Lu89a]).

```
Nmax  = MAX(Breite aller parallelen Pfade) + 1;
```

Dazu paßt die Vorstellung, daß jeder Pfad eines parallelen Bereichs einem virtuellen Prozeß zugeordnet wird. Die maximale Pfadbreite von vier in Abbildung 5.3, einem Ausschnitt aus der Gesamttabelle einer Prozessorkonfiguration, wird im vierten von insgesamt sieben parallelen Bereichen erreicht. Die vorderste Tabelle zeigt die Konfiguration für vier Prozessoren.

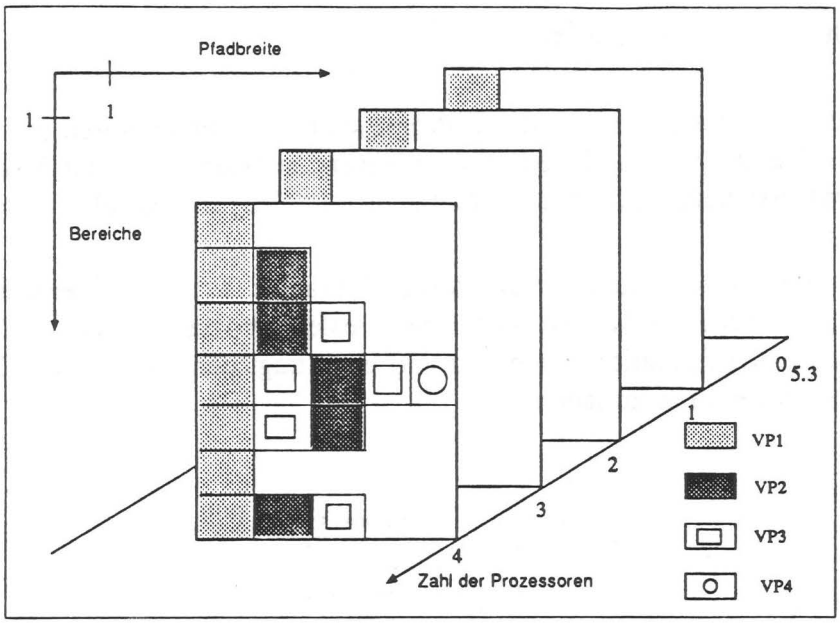

Abb. 5.3: virtuelle Prozesse

Wir betrachten im folgenden (Abbildungen 5.4 bis 5.5) als Beispiel die Zuordnung der Einheiten eines Programmschemas bis zum verteilten Programm auf mehreren Prozessoren.

```
Gegeben:                Seq. Hauptprogramm

Präprozessor            im paralleler Bereich 1 (PB1)
ordnet zu und              p1(x)      ->    Aktion1(1)
beschreibt:                p2(y,s)    ->    Aktion2(1)
                           p3(z)      ->    Aktion3(1)
                           p1(w)      ->    Aktion4(1)

                        im paralleler Bereich 2 (PB2)
                           p1(xx)     ->    Aktion1(2)
                           p2(yy,ss)  ->    Aktion2(2)

Gegeben:                Zuordnungen des Präprozessors

Konfigurator            Seq. Hauptprogramm ->  virtueller Prozeß1 (VP1);
ordnet zu und           Aktion1(1)     ->   virtueller Prozeß2 (VP2);
beschreibt:             Aktion2(1)     ->   virtueller Prozeß3 (VP3);
                        Aktion3(1)     ->   virtueller Prozeß4 (VP4);
                        Aktion4(1)     ->   virtueller Prozeß5 (VP5);

                        Aktion1(2)     ->   virtueller Prozeß2 (VP2);
                        Aktion2(2)     ->   virtueller Prozeß3 (VP3);
```

Abb. 5.4: Konfigurierung

5.2. Laden des verteilt ablaufenden Programms

Wir nehmen an, daß die initiale Bindung der E-Prozesse an reale Prozessoren (durch das Betriebssystem) aus Flexibilitäts- und Fehlertoleranzgründen (wenn z.B. der Multicomputer erweitert wird) erst beim Aufruf des parallelisierten Programms, also durch den Systemlader, erfolgt.

Mit den Daten der aktuell gültigen Prozessorkonfiguration lädt der Systemlader den initialen Prozeß des *E-Prozeß-Clusters*, auf einen realen Prozessor. Der geladene Prozeß besorgt sich die Konfigurationsdaten *.cfd* und beauftragt den Systemlader Kopien von sich auf weitere Prozessoren zu laden.

```
Systemlader:
                        E-Prozeß1   -> Prozessor#7
                        E-Prozeß2   -> Prozessor#3
                        E-Prozeß3   -> Prozessor#12
                        E-Prozeß4   -> Prozessor#4
```

Abb. 5.5: Verteilung durch Systemlader

Dabei kann je nach Vorgabe des Zielbetriebssystems entweder der initiale Prozeß mit Hilfe der Liste der aktiven Prozessoren die Zielprozessoren bestimmen, oder aber die Verteilung dem Systemlader überlassen. Entscheidend ist, daß die E-Prozesse eines Clusters nach dem Laden ihren Clustermitgliedern Botschaften senden können.

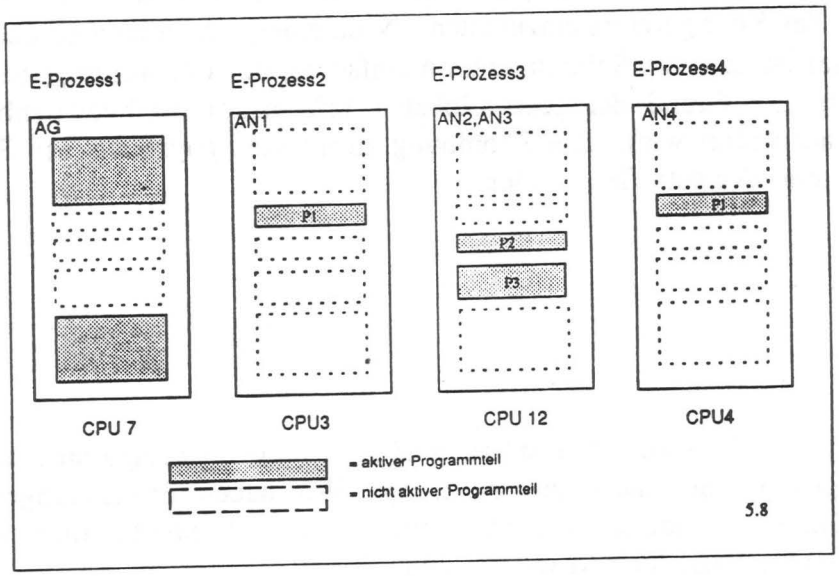

Abb. 5.6: Aktivierte Tasks im E-Prozeß-Cluster

5.3. Dynamisches Verteilen zur Laufzeit

Soll statt zur Ladezeit erst zur Laufzeit am Beginn eines parallelen Bereichs, eine dynamische, da vielleicht lastabhängige Zuteilung der Tasks an E-Prozesse erfolgen, so muß ein Laufzeitunterprogramm die Aufgabe des Systemdispatchers (Prozedurverteiler statt Jobverteiler) übernehmen (s. Abb. 5.7). Zusätzlich ist dieser Punkt im Kontrollfluß sogar geeignet, eine Rekonfiguration z.Bsp. nach Ausfall eines Prozessors zuzulassen.

Das Laufzeitunterprogramm startet auf einem Prozessor und startet dann die weiteren E-Prozesse des E-Prozeß-Clusters auf anderen Prozessoren. Der Vorteil dabei ist, daß die Parallelisierung mit den zur Laufzeit gültigen Daten aus dem Programm heraus gesteuert wird. Der Nachteil dabei ist der größere Overhead, durch den zusätzlichen nur einmal genutzten Code des verteilenden Laufzeitprogramms.

```
Beauftragung:
(zur Laufzeit)          Daten f. Aktion1 -> Prozessor#7
                        Daten f. Aktion2 -> Prozessor#3
                        Daten f. Aktion3 -> Prozessor#12
                        Daten f. Aktion4 -> Prozessor#6
```

Abb. 5.7: Verteilung zur Laufzeit

Zur Ausführungszeit erfragt ParBegin im *E-Prozeß* die aktuell gegebene Auftragsverteilung, d.h. die Abbildung der *E-Prozesse* auf die physikalisch vorhandenen, aktiven Prozessoren. Die Zuteilung der Prozessoren erfolgt beim Laden, denn erst dann ist bekannt, welche physikalischen Prozessoren verfügbar sind.

Beim Laden des E-Prozesses wird also vom Systemlader die aktuelle Zuordnung von virtuellen an reale Prozessoren vorgenommen. Aus Fehlertoleranzgründen erscheint es sinnvoll, die Zuordnung der Tasks für jeden parallelen Bereich während der Laufzeit nach den Vorgaben des Konfigurators einzuhalten. Nach einer Fehlerdiagnose und Ausfall eines E-Prozesses ist dann die Rekonfiguration einfacher, da vom Auftraggeber bzw. den Auftragnehmern auf Grund der prozesslokalen Information nachvollziehbar ist, welche Aktion wo ausgeführt wird. Die Zuordnung zu intakten physikalischen Prozessoren muß vom Systemlader getroffen werden.

6. Schlußbetrachtung

Aus einfachen Anforderungen zur Unterstützung eines parallelen Programms ergeben sich Folgeforderungen an die Laufzeitunterstützung. Wir haben Erweiterungen aufgezeigt, mit deren Hilfe sequentielle, blockorientierte, prozedurale Programmiersprachen aus Anwendersicht einfach parallelisiert werden können.

Die *einfache* Parallelität auf Prozedurebene wird in unserem Falle nicht durch eine neue Programmiersprache realisiert, sondern durch eine Kombination von Präprozessor, Konfigurator, Auftragsverteiler und erweiterte Laufzeitunterstützung erreicht.

Literatur

[Cam] Modula-2 Compiler
Computer Laboratory, Cambridge, England, k.J.

[Dal87] DalCin,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

[DLR88] DalCin,M.;Lutz,J.;Risse,Th. "Programmierung in Modula-2"
3.Aufl., B.G. Teubner Stuttgart 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

[Lu89a] Lutz,J.;Brause,R.;Dal Cin,M.;Philipp,Th. "Parallelisierungskonzept für ATTEMPTO-2" Interner Bericht 1/89 Fachbereich Informatik Universität Frankfurt

[Lu89b] Lutz,J. "Parallelisierung und Verteilung von Betriebssystemen am Beispiel von MINIX" Dissertation am Fachbereich Informatik Universität Frankfurt, in Bearbeitung

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

[TDI] The TDI Modula-Compiler Reference Manual
Modula-2 Software Ltd., Bristol, U.K. 1986

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

Anhang

A. Stackoffset bei Prozeduraufruf

Zum Auslesen des Parameterbereichs innerhalb des Aufrufstacks einer Prozedur ist es notwendig, dessen Lage relativ zu den lokalen Variablen zu kennen. Das folgende kleine Testprogramm dient dazu, den in *CallService* (s.Abb. 2.6) benötigten Offset einmal (je Modula-Implementierung) zu bestimmen.

Es werden beim Testprogramm folgende (nicht wesentliche) Annahmen gemacht:

- Stack wächst in Richtung kleiner Adressen
- Feldelemente mit kleinerem Index liegen bei kleineren Adressen
- Die zuletzt definierte lokale Variable einer Prozedur liegt an der kleinsten Adresse, d.h. auf der Spitze des nach kleinen Adressen wachsenden Stacks.

Dieses kleine Testprogramm funktioniert in etwa so:

Der Parameter von PROC1Body wird durch die Anweisung ptrParam^ := 5; auf dem Stack plaziert. Den Stackpointer bekommt man über ADR(LastLocalVar). Dann wird PROC1Body über eine Prozedurvariable so aufgerufen, als ob sie eine parameterlose Prozedur wäre, damit die zuvor künstlich erzeugten Parameter auf dem Stack nicht mit neuen Parametern überschrieben werden. Für falsche Werte von Offset findet das Programm statt der erwarteten '5' zufällige Werte.

```
MODULE M2TEST;

   (*  Testprogramm zur Validierung der Annahmen über Modula-2     *)
   (*  Compiler und zur Bestimmung systemspezifischer Konstanten.  *)

FROM SYSTEM IMPORT ADR, ADDRESS, WORD, TSIZE;
FROM InOut  IMPORT WriteCard, WriteString, WriteLn;

CONST
   StackWordSize = 2;

VAR
   PROCVAR   : PROC;
   Offset    : ADDRESS;
   indx      : CARDINAL;


PROCEDURE PROC1Body( Value : CARDINAL );
BEGIN
   IF Value = 5 THEN
      WriteString( 'Offset ermittelt.' ); WriteLn;
      WriteCard( Offset,5 ); WriteLn;
   END;
END PROC1Body;
```

```
PROCEDURE PrA;
VAR
   ptrParam     : POINTER TO CARDINAL;
   LastLocalVar : CARDINAL;
BEGIN
   PROCVAR    := PROC(PROC1Body);
   ptrParam   := ADR(LastLocalVar) - Offset;   (*  Annahme: Stack wchst in     *)
   ptrParam^  := 5;                            (*  Richtung kleinerer Adressen *)
   PROCVAR;
END PrA;


BEGIN  (* M2TEST *)
   FOR indx:=0 TO 32 BY StackWordSize DO
      Offset := ADDRESS( indx );
      PrA;
   END;
END M2TEST.
```

Abb. A.1: Offset Test

Daß der Offset für TDI-Modula gleich 0 ist bedeutet, daß die Prozedur Proc1Body ihre Parameter dort erwartet, wo sich die lokalen Variablen der aufrufenden Prozedur (PrA) befinden (dies ist inbesondere auch bei von C-Compilern erzeugtem Code der Fall). Für die Anwendung bedeutet dies, daß die Größe (TSIZE) des Typs der Variablen LastLocalVar so groß gewählt wird, daß dort die Argumente der künstlich aufzurufenden Prozedur Platz finden.

In Modula-2 läßt sich also eine Prozedur, die Parameter erwartet, auch (über Typkonvertierung und Prozedurvariable) so aufrufen, als ob sie eine parameterlose Prozedur wäre. Der Unterschied besteht nur darin, daß die angesprungene Prozedur statt ihrer Parameter nur Datenmüll (von früheren Aufrufen anderer Prozeduren) vorfindet. Das läßt sich ändern, indem man vor dem Aufruf die Parameter explizit (über Wertzuweisungen an geeignet berechnete Adressen) auf die richtigen Stackadressen kopiert.

Dieser oben vorgestellte Mechanismus ist kein von der Sprache Modula-2 abhängiges Verfahren, sondern kann auch mit anderen Hochsprachen (z.B. in C durch Zeiger) einfach realisiert werden. Dadurch ist das gesamte Konzept einfach in andere Sprachen und auch auf andere Systeme übertragbar.

B. Skizzen zur Implementation dynamischer Verteilung

ParEnd

Im statischen Fall lief ParEnd nur auf einem Auftraggeber ab. Im dynamischen Fall können auch Auftragnehmer ihrerseits als Auftraggeber diese Prozedur ausführen. Jetzt soll ParEnd bei den Auftragnehmern der nullten Stufe betrachtet werden. (Unter Auftragnehmern nullter Stufe werden all diejenigen E-Prozesse verstanden, die nach der Initialisierung in den Wartezustand gehen und auf Aufträge warten.) Die folgenden Programmfragmente setzen insbesondere die Prozeduren DoService(message) und RememberResultAndDispatch(message) voraus. Sie werden anschließend beschrieben. Wesentlich für dieses Konzept ist das Warten auf Ergebnisse und Aufträge in ParEnd.

```
TYPE
   tMessage = POINTER TO ARRAY [0..MaxMessLength] OF WORD;

   (*  Nachrichten enthalten ihren Typ (ServiceRequest oder Result),  *)
   (*  ihre Länge, eine Bereichsnummer (ParBeginParEndRegionNumber),  *)
   (*  Schachtelungstiefe, Aktionsnummer, Auftraggeber und -nehmer    *)

VAR
   CurrentProcess : POINTER TO PROCESS;


PROCEDURE ParEnd( ...; ParRegionNumber : CARDINAL; ... );
VAR
   mess : tMessage;
BEGIN
   EnterParEnd( ParRegionNumber );
   LOOP
      ReceiveAMessage( mess );
      IF MessageTyp(mess) = ServiceRequest THEN
         DoService(mess);
      ELSIF MessageTyp(mess) = Result THEN   (*  Möglicherweise das Ergebnis *)
         RememberResultAndDispatch(mess);    (*  eines höheren Par-Bereichs  *)
      END;
      IF AllResultsOfThisParRegionReceived() THEN
         Check1;
         Check2;
         UpdateResults;
         RETURN;
      END;
   END;
END ParEnd;
```

Abb. B.1: dynamisches ParEnd

Beim Start des parallelen Programms übernimmt der erste, sequentielle Bereich des gestarteten Programms die Initialisierung aller gemeinsamen globalen Variablen. Nachdem alle Prozesse des E-Prozeß-Clusters existieren, folgt die Unterscheidung in Auftraggeber und Auftragnehmer der nullten Stufe:

```
BEGIN  (*  Initialisierung des Hauptprogramms  *)

   ...                                  (*  Erster sequentieller Bereich      *)
   (* fork *)
   ...                                  (*  Zweiter sequentieller Bereich     *)


   IF WhoAmI() # Auftraggeber THEN  (*  Auftragnehmer nullter Stufe        *)
      LOOP
         ReceiveAMEssage( mess );
         IF MessageTyp(mess) = ServiceRequest THEN
            DoService( mess );
         ELSIF MessageTyp(mess) = TerminateLevel0Server THEN
            EXIT (* LOOP *);
         ELSE
            Error(...);
         END;
      END;
   ELSE                                 (*  Auftraggeber nullter Stufe        *)
      DoNormalSequentialWork;
      TerminateAllLevel0Server;         (*  Durch Versenden einer Botschaft   *)
   END;                                 (*  des Typs TerminateAllLevel0Server *)
END Hauptprogramm.
```

Abb. B.2: Initialisierung im Hauptmodul

Spezifikation von DoService(message)

Im vorigen Abschnitt haben wir den Aufbau des Stacks gesehen. Diese Kenntnis nutzen wir nun aus, um die Implementierung von DoService als Coroutine zu skizzieren (Abb. B.3). Wir wählen die Darstellung einer Coroutine, da sie einen eigenen Stack besitzt und nebenläufig ausgeführt werden kann. Jeder Service soll mittels eines Leichtgewichtsprozesses (Coroutine) erbracht werden, der einem Pool entnommen wird (s.Abb. B.4). Ist der Pool der voneinander unabhängigen Leichtgewichtsprozesse erschöpft, wird vom Auftragnehmer kein weiterer Auftrag angenommen, sondern eine negative Rückantwort gegeben. Um jetzt Deadlocks zu vermeiden, dürfen parallele Aufträge nur noch dann vergeben werden, wenn zurückgewiesene Aufträge an andere E-Prozesse notfalls durch lokale Leichtgewichtsprozesse ausgeführt werden können.

```
VAR
   GlobalMessage : tMessage;

PROCEDURE DoService( mess : tMessage );
VAR
   Current,
   NewProcess : POINTER TO PROCESS;
BEGIN
   MakeANewServiceProcess( NewProcess );
   GlobalMessage := mess;
   Current := CurrentProcess;
   CurrentProcess := NewProcess;
   TRANSFER( Current^,NewProcess^ );
END DoService;
```

Abb. B.3: DoService

Service-Pool

Für alle Service-Prozesse wird beim Programmstart eine Tabelle von Service-Records initialisiert, die als Pool für die einzelnen Service-Aufrufe dient. Ein Record aus dem Pool wird mit *MakeANewServiceProcess* (s.Abb. B.4) entnommen.

```
TYPE
   tService = RECORD
                 RecordInUse,                    (*  Record verfügbar?      *)
                 AwaitingResults   : BOOLEAN;    (*  Austehende Resulate?   *)
                 ProcessDescriptor : PROCESS;

                 (*  Folgende Einträge sind relevant, falls der Service    *)
                 (*  Klient einer ParEnd() -Prozedur ist                   *)

                 ParRegionNumber,
                 DynamicRegionId   : CARDINAL;
                 ResultsToGet,
                 ResultsGot        : CARDINAL;
                 Results           : ARRAY [0..MaxResult] OF tMessage;
              END;


VAR
   ServiceTable               : ARRAY [0..Max] OF tService;
   GlobalDynamicRegionCounter : CARDINAL;


PROCEDURE MakeANewServiceProcess( NewProcess : POINTER TO PROCESS );
VAR
   p : POINTER TO tService;
BEGIN
   p := FindAnUnusedRecordInTheServiceTable();
   WITH p^ DO
      RecordInUse     := TRUE;
      AwaitingResults := FALSE;
      NEWPROCESS( Service, ... , ... ProcessDescriptor );
      NewProcess := ADR(ProcessDescriptor);
   END;
END MakeANEwServiceProcess;
```

Abb. B.4: Make a new Service-Process

Am Ende eines Bereichs müssen die Botschaften erwartet werden und je nach Botschaftstyp muß unterschiedlich fortgefahren werden (s. Abb.B.5).

```
PROCEDURE RememberResultAndDisptach( mess : tMEssage );
VAR
   ptrService : POINTER TO tService;
   Current    : POINTER TO PROCESS;
```

```
BEGIN
   ptrService := FindTheServiceTheResultBelongsTo( mess );

   (*  Zugriffsschlüssel sind ParRegionNumber und DynamicRegionId  *)

   WITH ptrService^ DO
      Results[AktionsNummer(mess)] := mess;    (*  Botschaft puffern, bis  *)
      INC( ResultsGot );
      IF ResultsGot = ResultsToGet THEN        (*  alle eingetroffen sind  *)
         AwaitingResults := FALSE;
         Current         := CurrentProcess;
         CurrentProcess  := ADR(ProcessDescriptor);
         IF Current^ # ProcessDescriptor THEN
            TRANSFER( Current^,ProcessDescriptor );
         END;
      END;
   END (* WITH *);
END RememberResultAndDispatch;
```

Abb. B.5: Remember Results and Dispatch

Terminierung eines Services

In *EnterParEnd* (s.Abb B.6) wird das dynamische *ParEnd* initialisiert, die dynamische Bereichsidentifikation (region ID) festgehalten und erhöht. Ist ein Service erbracht, dann wird der Service-Record mit *TerminateMeAndTransferControlToAnotherProcess* wieder freigeben und eine andere Coroutine aktiviert.

```
PROCEDURE EnterParEnd( ParRegionNumber : CARDINAL );
BEGIN
   WITH CurrentProcess^ DO
      ResultsToGet    := LookupTableForResultsToGet( ParRegionNumber );
      ResultsGot      := 0;
      AwaitingResults := TRUE;
      DynamicRegionId := GlobalDynamicRegionId;
      INC( GlobalDynamicRegionId );
   END;
END EnterParEnd;


PROCEDURE TerminateMeAndTransferControlToAnotherProcess;
VAR
   ptrProcess : POINTER TO PROCESS;
   ptrService : POINTER TO tService;
BEGIN
   CurrentProcess^.RecordInUse := FALSE;
   ptrProcess     := CurrentProcess;
   ptrService     := FindAUsedServiceRecord();

   (*  Keine Rückkehr zum Hauptprogarmm, solange Prozesse existieren  *)

   CurrentProcess := ADR(ptrProcess^.ProcessDescriptor);
   TRANSFER( ptrProcess^,ptrService^.ProcessDescriptor );
END TerminateMeAndTransferToAnotherProcess;
```

```
VAR
   indx          : CARDINAL;
   ServiceTable : ARRAY [0..MaxServiceTableEntries] OF tService;

BEGIN  (* Initialisierung *)
   GlovbalDynamicRegionId := 0;
   FOR indx:=1 TO MaxServicetableEntries DO
      ServiceTable[indx].RecordInUse := FALSE;
   END;
   WITH ServiceTable[0] DO              (*  Hauptprogramm  *)
      RecordInUse     := TRUE;
      AwaitingResults := FALSE;
   END;
END Modul.
```

Abb. B.6: Paralleler Service

INTERNE BERICHTE AM FACHBEREICH INFORMATIK, UNIV. FRANKFURT

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 u.a.: 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

2/89 Lutz, Joachim [u.a.]: Die Erweiterungen der ATTEMPTO-2 Laufzeitbibliothek

3/89 Kemp, Rainer: A One-to-one Correspondence between Two Classes of Ordered Trees