Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

Petra Hofstedt
Technische Universität Dresden
hofstedt@inf.tu-dresden.de

Kurzfassung

Algorithmische Skelette sind polymorphe Funktionen höherer Ordnung, die allgemeine Parallelisierungsmuster repräsentieren und parallel implementiert sind. Skelette können zur Integration paralleler Anwendungen in sequentielle Sprachen genutzt werden. Auf diese Weise werden die Ausdrucksstärke und Flexibilität der sequentiellen Basissprache mit der Effizienz paralleler Algorithmen für spezielle Anwendungsfälle kombiniert.
In dieser Arbeit wird zunächst das Konzept der Skelette mit Hilfe typischer Beispiele daten- und taskparalleler Skelette erläutert.
Die Einbettung eines taskparallelen Skeletts in eine funktionale Sprache und die Anwendung der erweiterten Sprache auf ein spezielles Problem werden anhand des Branch-and-Bound-Verfahrens gezeigt.
Es wurden vergleichende Messungen für Maschinenbelegungsprobleme verschiedener Größen und auf unterschiedlichen Netzwerkgrößen auf einem Parallelrechner Parsytec MultiCluster 2 mit 64 frei verschaltbaren Transputern des Typs T800 - 20 MHz und einem GigaCluster GCel1024 mit 1024 Transputern T805 - 30 MHz durchgeführt und ausgewertet.

Inhaltsverzeichnis

1 Einführung

Deklarative Sprachen unterstützen die Entwicklung zuverlässiger Software. Sie stellen Abstraktionsmittel bereit, die direkt auf formalen Kalkülen basieren. Dadurch wird die Entwicklung und Ausnutzung von Programmtransformations-, -analyse- und -verifikationstools erleichtert.

Die Mehrheit paralleler Programme wurde bisher mit Hilfe von imperativen Sprachen geschrieben. Viele dieser Sprachen zwingen den Programmierer, low-level-Konstrukte zur Synchronisation und Kommunikation zu verwenden. Bestehende imperative Programmiersprachen können zwar zur parallelen Behandlung von Problemen mit regulären Strukturen, wie sie z.B. aus der Bildverarbeitung bekannt sind, eingesetzt werden. Die Programmierung unregelmäßiger Probleme, wie z.B. bei adaptiven numerischen Verfahren, ist hingegen nur mit hohem Aufwand (dynamische Lastverteilung usw.) möglich. Funktionale Sprachen mit sogenannten algorithmischen Skeletten bieten hier eine Alternative.

2 Algorithmische Skelette

Algorithmische Skelette sind polymorphe Funktionen höherer Ordnung, die allgemeine Parallelisierungsmuster repräsentieren und parallel implementiert sind. Skelette können in sequentielle Sprachen eingebettet werden und sind dann einzige Quelle von Parallelität im Programm. Parallele Implementationsdetails sind innerhalb der Skelette verborgen. Die Skelette selbst sind in einer niederen Programmiersprache implementiert und daher effizient. Auf diese Weise werden die Ausdrucksstärke und Flexibilität der sequentiellen Basissprache mit der Effizienz von Algorithmen für spezielle Anwendungsfälle kombiniert.

Da die Definition von Skeletten nur von der Struktur des Problems und nicht von speziellen Datentypen abhängig sein soll, ist ein polymorphes Typsystem notwendig.

Implementierte Skelette kann man als externe Funktionen dem Nutzer direkt bereitstellen oder zur Erweiterung der verwendeten Basissprache z.B. um Constraint-Systeme verwenden.

In Abhängigkeit der ausgenutzten Parallelität unterscheidet man daten- und taskparallele Skelette.

2.1 Datenparallele Skelette

Datenparallele Skelette agieren auf verteilten Datenstrukturen und führen gleichzeitig die gleiche Operation auf den Elementen dieser Datenstruktur aus. Die Verteilung der Elemente der Datenstruktur auf die verschiedenen Prozessorelemente wird vom Nutzer explizit angegeben.

Typische Beispiele für datenparallele Skelette sind map und fold auf verteilten Arrays.

Beispiel 2.1 In einer funktionalen Sprache soll es folgende datenparallele Skelette geben:

- `create_array x y xv yv w` erzeugt ein verteiltes Array in Abhängigkeit von 5 Argumenten: der Dimension des Arrays in x-Richtung `x`, der Dimension des Arrays in y-Richtung `y`, der Ausdehnung der Netztopologie in x-Richtung `xv`, der Ausdehnung der Topologie in y-Richtung `yv` und einem Initialwert `w`, der für jedes der Elemente des Arrays gilt.

  ```
  IMPORT create_array:
   INT → INT → INT → INT → INT → ARRAY;
  ```

- `update_array a x y w` bildet aus dem verteilten Array `a` ein neues verteiltes Array, wobei an der durch `x` und `y` spezifizierten Position des neuen Arrays jetzt der Wert `w` steht.

  ```
  IMPORT update_array:
   ARRAY → INT → INT → INT → ARRAY;
  ```

- `map_array fun a` wendet die Funktion `fun` auf das Array `a` an und bildet dabei ein neues Array. Die Funktion `fun` bildet dabei in Abhängigkeit von der Position und dem Wert eines Arrayelements den neuen Wert dieses Elements.

  ```
  IMPORT map_array:
   (INT → INT → INT → INT) → ARRAY → ARRAY;
  ```

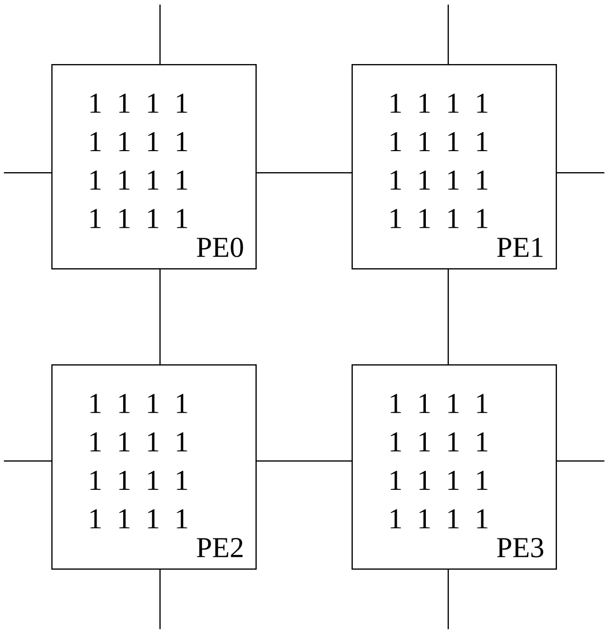
Außerdem wird eine Funktion `bedingt_mal_zwei` definiert, die in Abhängigkeit der Position eines Elements im Array den ursprünglichen Wert auf 0 setzt, wenn sich das Element auf der Diagonalen des Arrays befindet, und sonst verdoppelt:

```
VAL bedingt_mal_zwei: INT → INT → INT → INT;
x y v ⇒ IF x=y THEN 0 ELSE v*2 FI;
```

Dann kann man die Kreation eines verteilten Arrays aufrufen:

```
EVAL create_array 8 8 2 2 1 .
```

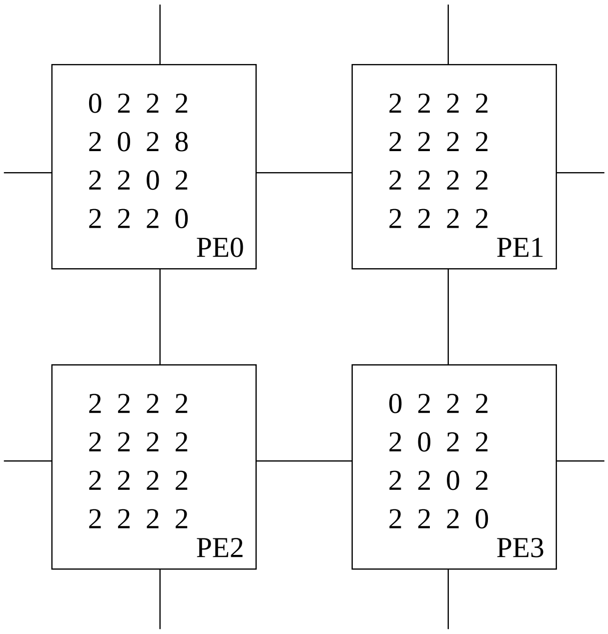
und erzeugt damit ein folgendermaßen verteiltes Array:



Verändert man den Wert des Elements an Postion `3 1` in diesem Array zu 4 und wendet dann `bedingt_mal_zwei` auf das Gesamtarray an

```
EVAL map_array bedingt_mal_zwei
 (update_array (create_array 8 8 2 2 1) 3 1 4) .
```

so erhält man folgendes Resultatarray:



2.2 Taskparallele Skelette

Taskparallele Skelette generieren eine Reihe von Prozessen, die parallel abgearbeitet werden können. Die zu verteilenden Prozesse müssen nicht von vornherein bekannt sein und können dynamisch erzeugt werden. Die verteilte Datenstruktur wird vom System verwaltet und ist für den Nutzer nicht sichtbar.

Beispiele für taskparallele Skelette sind farm, pipe, divide&conquer und branch-and-bound.

Beispiel 2.2 Ein Skelett hat eine deklarative Bedeutung, die durch seine Definition in einer funktionalen Sprache angegeben werden kann. Diese Bedeutung ist unabhängig von der speziellen Implementierung des Skeletts. In einer funktionalen Sprache kann man divide&conquer wie folgt definieren:

```
VAL divide_conquer:
```
$(\alpha \to \beta) \to (\alpha \to BOOL) \to ([\beta] \to \beta) \to (\alpha \to [\alpha]) \to \alpha \to \beta$;

```
solve is_trivial combine divide arg ⇒
    IF (is_trivial arg) THEN (solve arg)
    ELSE combine (map
      (divide_conquer solve is_trivial combine divide)
       (divide arg))
    FI;
```

Entsprechend dem divide&conquer-Berechnungsmuster wird zunächst getestet, ob das Problem trivial ist, und gegebenenfalls direkt gelöst. Sonst wird das Problem in eine Reihe von Unterproblemen aufgeteilt und `divide_conquer` auf jedes dieser Unterprobleme angewendet. Am Ende werden die Teillösungen zur Gesamtlösung zusammengefügt.

`divide_conquer` hat dabei einen polymorphen Typ und ist eine Funktion höherer Ordnung. Verschiedene Algorithmen, die die divide&conquer-Struktur haben, können durch Angabe verschiedener Argumentfunktionen implementiert werden. Ein Beispiel ist der quicksort-Algorithmus:

```
VAL quicksort: [INT]→[INT];
list ⇒ divide_conquer id empty_one comb_list div_list list;
```

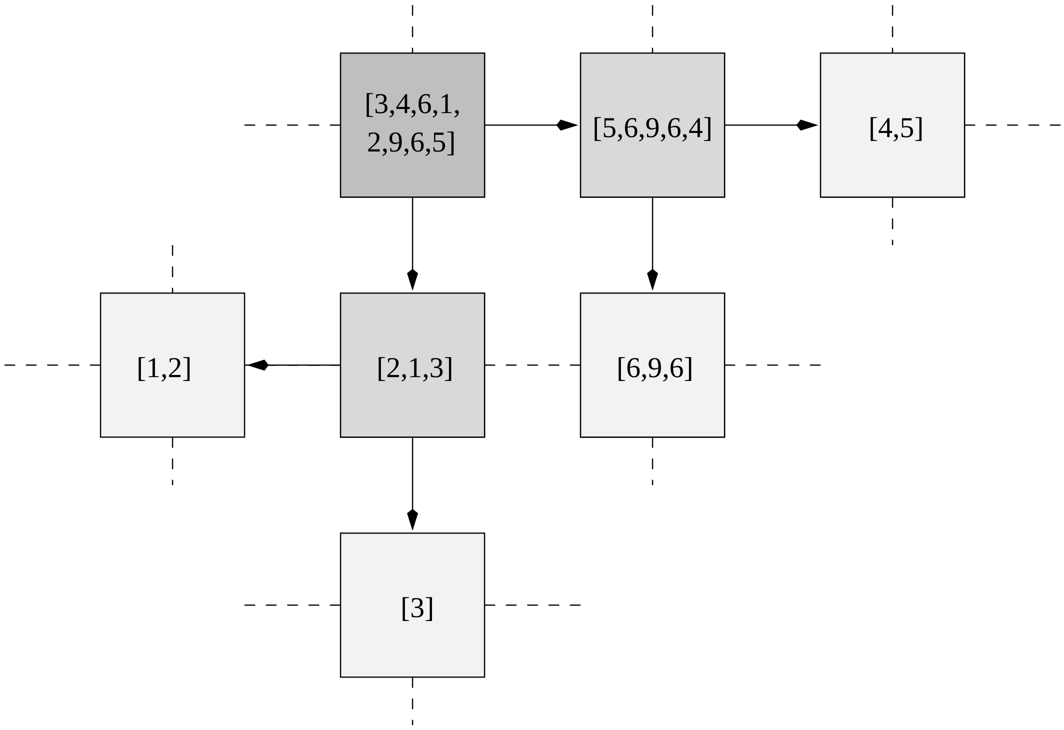
Stellt die funktionale Sprache ein `divide_conquer`-Skelett zur Verfügung, so braucht der Nutzer die obige Funktionsdefinition für divide&conquer nicht mehr anzugeben, sondern das Skelett nur noch zu importieren:

`IMPORT divide_conquer:`
$(\alpha \to \beta) \to (\alpha \to BOOL) \to ([\beta] \to \beta) \to (\alpha \to [\alpha]) \to \alpha \to \beta;$

Man definiert `quicksort`, wie oben angegeben, und kann dann die Funktion anwenden:

```
EVAL quicksort [3,4,6,1,2,9,6,5] .
```

In diesem Beispiel wird das Problem wiederholt in Teilprobleme aufgespalten und diese an verschiedene Prozessoren verteilt und von diesen gelöst.



3 Branch-and-Bound

3.1 Grundlagen

Branch-and-Bound-Verfahren sind systematische Probierverfahren zur Lösung diskreter Optimierungsprobleme, wobei ein großer Teil der theoretisch möglichen Lösungen durch Abschätzungen als nichtoptimal ausgeschlossen werden kann. Man geht von einer Menge $X = \{x_1, \dots, x_n\}$ von Variablen aus, denen jeweils endlich viele diskrete Werte zugeordnet werden können. Ziel ist es, den Variablen solche Werte aus dem jeweiligen Wertevorrat zuzuordnen, daß eine bestimmte Funktion $z = z(x_1, \dots, x_n)$, die sogenannte Zielfunktion, einen Minimal- bzw. Maximalwert annimmt und bestimmte Nebenbedingungen erfüllt sind. Man nennt solche Probleme auch kombinatorische Optimierungsprobleme.

Typische Anwendungsbeispiele sind Maschinenbelegungsprobleme, das knapsack problem oder Rucksackproblem, das häufig als Teilproblem in größeren Optimierungsproblemen auftritt, das traveling salesman problem oder Rundreiseproblem und andere.

Branch-and-Bound stellt eine Suche in ODER-Bäumen dar. Aus einem gegebenen Ausgangsproblem werden zunächst entsprechend einer Verzweigungsregel (`branch`) paarweise elementfremde Subprobleme gebildet. Für jedes der neu entstandenen Subprobleme wird eine Abschätzung (`bound`) berechnet und aufgrund dieser Abschätzung ein im folgenden zu verzweigendes Problem ausgewählt (`select`) und ebenfalls verzweigt. Kann das ausgewählte Problem nicht weiter in Subprobleme zerlegt werden, so stellt seine Lösung, falls eine existiert, eine Optimallösung dar. Subprobleme mit nichtoptimalen oder unzulässigen Variablenbelegungen können im Verlauf der Berechnung aus dem Baum „herausgeschnitten“ werden (`elimination`).

Branch-and-Bound benötigt also eine Menge von Regeln:

- zum Verzweigen eines Problems in Subprobleme (`branch`),
- zur Bestimmung einer unteren bzw. oberen Schranke einer Optimallösung eines Subproblems (`bound`),
- zur Auswahl des im folgenden zu verzweigenden Problems (`select`),
- zur Erkennung von Problemen, die nicht zu zulässigen oder optimalen Lösungen führen können (`elimination`).

Im folgenden werden nur Minimierungsprobleme betrachtet. Sei P_0 ein zu lösendes Minimierungsproblem. P_0 wird entsprechend der `branch`-Regel in Subprobleme dekomponiert (verzweigt). Der endliche Baum $B = (P, A)$, dessen Knotenmenge P und dessen Kantenmenge A ist, repräsentiert diese Dekomposition. Er wird *Problembaum* (Abb.1) genannt. P_0 ist die Wurzel des Baums. Wird ein Subproblem P_j durch die Dekomposition des Problems P_i generiert, so gibt es eine Kante von P_i nach P_j, d.h. $(P_i, P_j) \in A$. Jedem Knoten aus P ist damit eineindeutig ein Subproblem zugeordnet.

Sei $f(P_i)$ eine Optimallösung für ein Subproblem P_i, und P_i werde dekomponiert in die Subprobleme $P_1, \dots, P_k$. Dann gilt:

$$f(P_i) = min_{j=1,\dots,k}\ f(P_j)$$

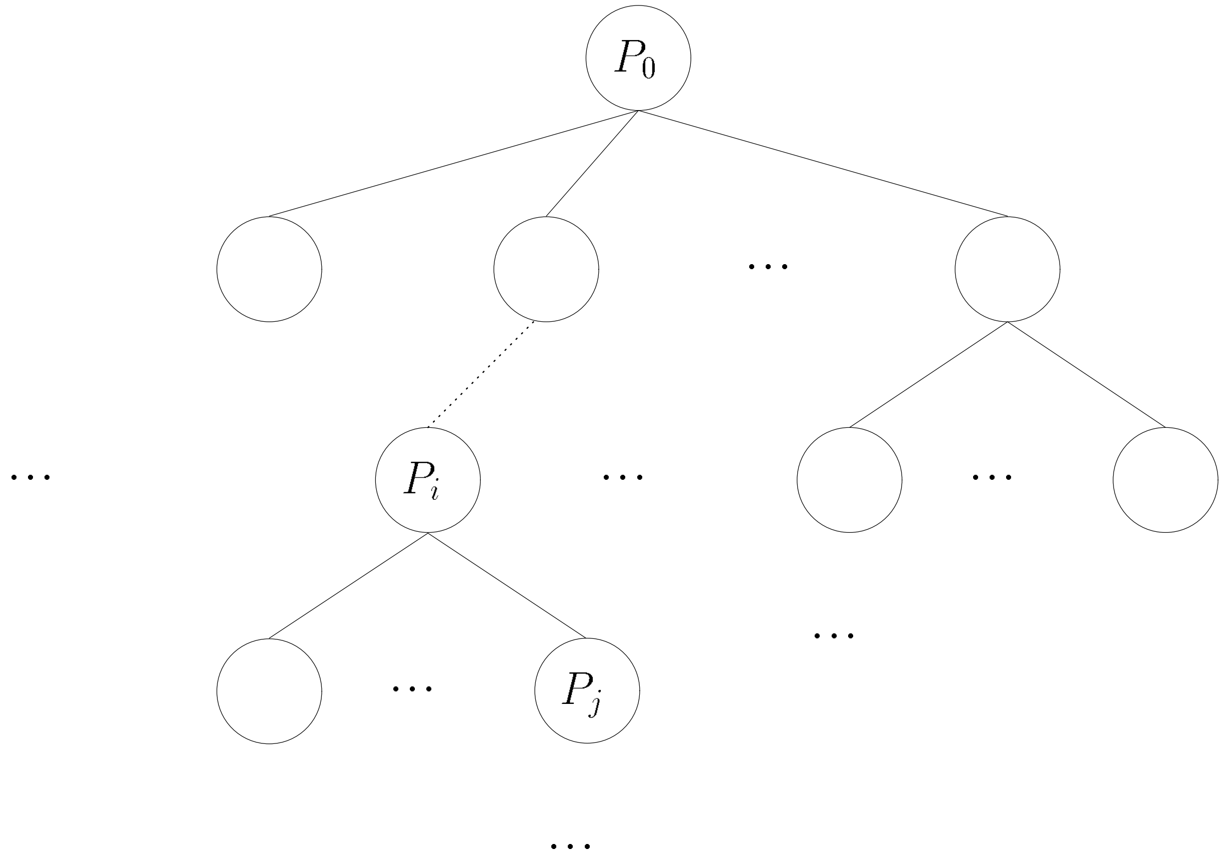


Abbildung 1: Problembaum

Es sei $g(P_i)$ die mit Hilfe der `bound`-Regel berechnete untere Schranke für den Zielfunktionswert einer Optimallösung des Subproblems P_i. T sei die Menge der ohne Dekomposition lösbaren Subprobleme. Folgende Eigenschaften gelten für g:

$$g(P_i) \leq f(P_i), \quad P_i \in P \tag{1}$$
$$g(P_i) = f(P_i), \quad P_i \in T \tag{2}$$
$$g(P_i) \leq g(P_j), \quad (P_i, P_j) \in A \tag{3}$$

Die Eigenschaft (1) besagt, daß g eine untere Schranke von f ist, die Eigenschaft (2), daß g und f übereinstimmen, wenn P_i ohne Dekomposition lösbar ist, und die Eigenschaft (3), daß g mit zunehmender Tiefe im Baum nicht sinkt.

3.2 Paralleles Branch-and-Bound

Der Hauptunterschied zwischen parallelen und sequentiellen Branch-and-Bound-Algorithmen besteht in der Behandlung und Nutzung des durch den Algorithmus generierten Wissens. Dazu zählen zum einen aus Problemen durch Dekomposition erzeugte Subprobleme und andererseits das Wissen um lokale und globale Optima.

In [10] gibt Trienekens eine Klassifikation paralleler Branch-and-Bound-Algorithmen anhand verschiedener Kriterien an und ordnet verschiedene solche Algorithmen aus der Literatur ensprechend dieser Kriterien ein.

Klassifikation nach Trienekens:

knowledge sharing	global/local knowledge base complete/partial knowledge base update strategy
knowledge use	access strategy reaction strategy
dividing the work	units of work load balancing strategy
synchronicity	synchronicity of each process

basic rules	`branch`
	`bound`
	`select`
	`elimination`

Wissensbasen können zunächst anhand der auf sie zugreifenden Prozesse und dem in ihnen gespeicherten Wissen klassifiziert werden. Wissensbasen können von allen Prozessen oder nur einer Teilmenge der Prozesse gelesen und/oder geschrieben werden (*global* vs. *local knowledge base*), sie können alles durch alle Prozesse generierte Wissen enthalten (*complete knowledge base*) oder nur einen Teil davon (*partial knowledge base*). Diese Eigenschaften wiederum können beliebig kombiniert auftreten (z.B. local partial knowledge base). Jede der Kombinationen hat ihre Vor- und Nachteile. Die *Aktualisierungs-Strategie* (update strategy) bezeichnet, wann und in welchem Maße eine Wissensbasis durch neu generiertes Wissen aktualisiert wird.

Die Nutzung des Wissens kann durch die *Zugriffs-Strategie* (access strategy) und die *Reaktions-Strategie* (reaction strategy) näher bestimmt werden. Die Zugriffs-Strategie legt fest, auf welche Wissensbasis von einem Prozeß, der im Ablauf seiner

Berechnungen eine Entscheidung treffen muß, zugegriffen werden kann und wann. Die Reaktions-Strategie besagt, wie ein Prozeß von einer Aktualisierung einer Wissensbasis informiert wird und mit welchen Aktionen der Prozeß darauf reagiert.

Eine weitere Möglichkeit zu klassifizieren liegt in der Art und Weise der (Ver-)Teilung der Arbeit. Eine *Arbeitseinheit* (unit of work) kann in der Auswahl und dem Verzweigen eines Subproblems oder in der vollständigen Lösung eines Subproblems bestehen oder auch von beliebiger anderer Größe gewählt werden. Somit kann nach der Größe der Arbeitseinheiten unterschieden werden. Damit eng verbunden ist die *Lastbalancierungs-Strategie* (load balancing strategy), die ebenfalls ein Kriterium darstellt.

Man kann parallele Branch-and-Bound-Algorithmen danach unterscheiden, ob sie nach Beendigung einer Arbeitseinheit und Beginn einer neuen warten, bis alle anderen Prozesse ebenfalls ihre aktuelle Arbeitseinheit vollständig bearbeitet haben (*synchron*), oder unabhängig von diesen mit einer weiteren Arbeitseinheit fortfahren (*asynchron*).

Letztendlich ist ein wichtiges Kriterium durch die *Basis-Regeln* (basic rules) `branch, bound, select` und `elimination` gegeben.

Die Kriterien bedingen sich gegenseitig. So sind Synchronität der Arbeitsweise und Aktualisierungs-, Zugriffs- und Reaktions-Strategie eng miteinander verbunden. Die Kriterien sind von der den Implementierungen unterliegenden Maschinenarchitektur und ebenso von den speziellen Anwendungen der Algorithmen (spezifiziert durch `branch, bound, select` und `elimination`) abhängig. So eignen sich globale Wissensbasen für Architekturen mit gemeinsamem Speicher, während partielle lokale Wissensbasen verbunden mit passenden Aktualisierungs-, Zugriffs- und Reaktions-Strategien typisch für Architekturen mit verteiltem Speicher sind.

3.3 Maschinenbelegungsprobleme

Eine typische Anwendung des Branch-and-Bound-Verfahrens stellen Maschinenbelegungsprobleme dar.

Ausgangspunkt sind hierbei eine Anzahl von Produkten $p_1, \ldots, p_m$, die eine Reihe von Maschinen $M_1, \ldots, M_n$ durchlaufen, sowie die Bearbeitungszeiten der einzelnen Produkte je Maschine und die Reihenfolge, in der die Produkte die Maschinen passieren. Hierbei sollen alle Produkte jeweils alle Maschinen in einer fest vorgegebenen Reihenfolge durchlaufen. $t_{i,j}$ sei die Zeit, in der Produkt p_j auf Maschine M_i bearbeitet wird.

Gesucht ist die Reihenfolge, in der die Produkte die Maschinen passieren, wobei die Gesamtbearbeitungszeit t bis zur Fertigstellung aller Produkte minimal sein soll:

$$min\ t \ = \Sigma_{j=1}^{m} t_{n,j} + t_{n,s}$$

dabei ist $t_{n,s}$ die Gesamtstillstandszeit der Maschine M_n.

Um ein Maschinenbelegungsproblem mittels Branch-and-Bound zu lösen, benötigt man die Regeln `branch, bound, select` und `elimination`.

Es sei L eine Produktliste, bestehend aus Indizes der Produkte $p_1, \ldots, p_m$, und $\tau = 0, 1, \ldots$ bezeichne die Tiefe eines (Sub-)Problems im Problembaum. Zur Vereinfachung der Darstellung wird in der folgenden Regel `branch` und im späteren Beispiel 4.1 die Schreibweise $P_{(L,\tau)} = (x_1, \ldots, x_n)$ verwendet. Dabei sind $P_{(L,\tau)}$ das durch L und τ gekennzeichnete (Sub-)Problem und $(x_1, \ldots, x_n)$ zugehörige Zeitangaben für die Maschinen $M_1, \ldots, M_n$.
Ferner bezeichnet "[l:L]" die durch Konkatenation des Elements l und der Liste L entstehende Liste; "[]" steht für die leere Liste. "[1,2,3]" ist eine Abkürzung für "[1:[2:[3:[]]]]".

Die im folgenden angegebenen Regeln sind nicht als Beschreibung eines Lösungsalgorithmus zu verstehen.

- `branch`

  Ein Problem $P_{(L,\tau)} = (x_1, \ldots, x_n)$ wird zerlegt in eine endliche Anzahl von Subproblemen $P_{([l:L],\tau+1)}$ mit $l \in \{\{1, \ldots m\} - \{\lambda \mid \lambda \in L\}\}$, wobei gilt $P_{([l:L],\tau+1)} = (x'_1, \ldots, x'_n)$ mit $x'_1 = x_1 + t_{1,l}$ und $x'_i = max(x'_{i-1}, x_i) + t_{i,l}$, $i = 2, \ldots, n$.

- `bound`

  $U(P_{([l:L],\tau)})$ bezeichne die untere Schranke des Subproblems $P_{([l:L],\tau)}$. Dann gilt $U(P_{([l:L],\tau)}) = U(P_{(L,\tau-1)}) + ((x'_n - x_n) - t_{n,l})$, $\tau = 1, 2, \ldots$

- `select`

  Wähle ein Problem $P_{(.,.)}$ mit minimaler unterer Schranke aus. Gibt es mehrere Probleme mit gleicher minimaler unterer Schranke, dann wähle unter diesen das dem Knoten in maximaler Baumtiefe zugeordnete Problem:

  Sei Π die Menge aller aktiven Knoten im Problembaum,
  $U(P_{(L_1,\tau_1)}) = \ldots = U(P_{(L_k,\tau_k)}) = min\{U(P_{(L,\tau)}) \mid P_{(L,\tau)} \in \Pi\}$ und
  $\tau_{k^\star} = max(\tau_1, \ldots, \tau_k)$,
  dann wähle ein Problem $P_{(L_{k^\star},\tau_{k^\star})}$.

- `elimination`

  Lösche alle die Subprobleme $P_{(L,\tau)} \in \Pi$, für die gilt
  $U(P_{(L,\tau)}) \geq U_{\text{opt}}$, wobei U_{opt} eine lokale Optimallösung mit dem aktuell kleinsten Zielfunktionswert ist.
  Alle im Problembaum verbleibenden Knoten heißen aktiv.

4 Das Branch-and-Bound-Skelett in einer funktionalen Sprache

4.1 Die Sprache DFS

Ausgangspunkt ist die datenparallele funktionale Sprache DFS ([8]). DFS ist eine experimentelle Programmiersprache zur Anwendung auf Parallelrechnern. Die Spra-

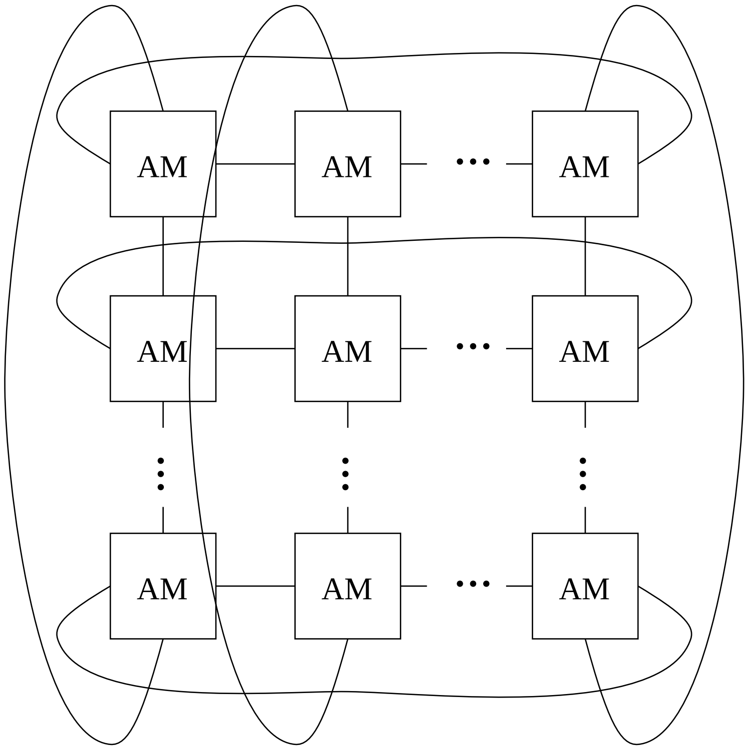


Abbildung 2: Parallele abstrakte Maschine

che ist strikt und wertet dementsprechend DFS-Programme nach der call-by-value-Strategie aus. DFS enthält bereits datenparallele Skelette für verteilte Arrays. DFS hat ein monomorphes Typsystem. Die Syntax der Sprache ist im Anhang angegeben.

Die Übersetzung eines korrekt getypten DFS-Programms in abstrakten Maschinencode erfolgt nach dem in [8] angegebenen Übersetzungsschema. Nach verschiedenen Optimierungen wird der optimierte Zwischencode in Transputer-Assembler übersetzt. Als letzter Schritt erfolgt die Übersetzung des Transputer-Assembler-Codes durch den Parix-Assembler.

4.2 Die abstrakte Maschine

Eine parallele abstrakte Maschine (PAM) stellt die Laufzeitumgebung für die funktionale Sprache dar. Diese parallele abstrakte Maschine besteht aus parallel arbeitenden abstrakten Maschinen, von denen jede als autonome Einheit auf je einem Prozessor arbeitet (Abb.2).

Jede einzelne der abstrakten Maschinen besteht aus zwei logischen Einheiten: der Nachrichtenverwaltungseinheit, die ankommende Nachrichten verarbeitet, und der Reduktionseinheit, die die eigentliche Berechnung durchführt (Abb.3).

Während der Initialisierungsphase baut jede der abstrakten Maschinen den lokalen Teil der virtuellen Topologie auf, berechnet ihre Routingtabelle und führt weitere Initialisierungen aus. Jede abstrakte Maschine lädt das Programm und begibt sich dann in einen Wartezustand, wo sie auf eingehende Nachrichten, die eine Skelettabarbeitung initiieren können, wartet. Einen ausgezeichneten Prozessor

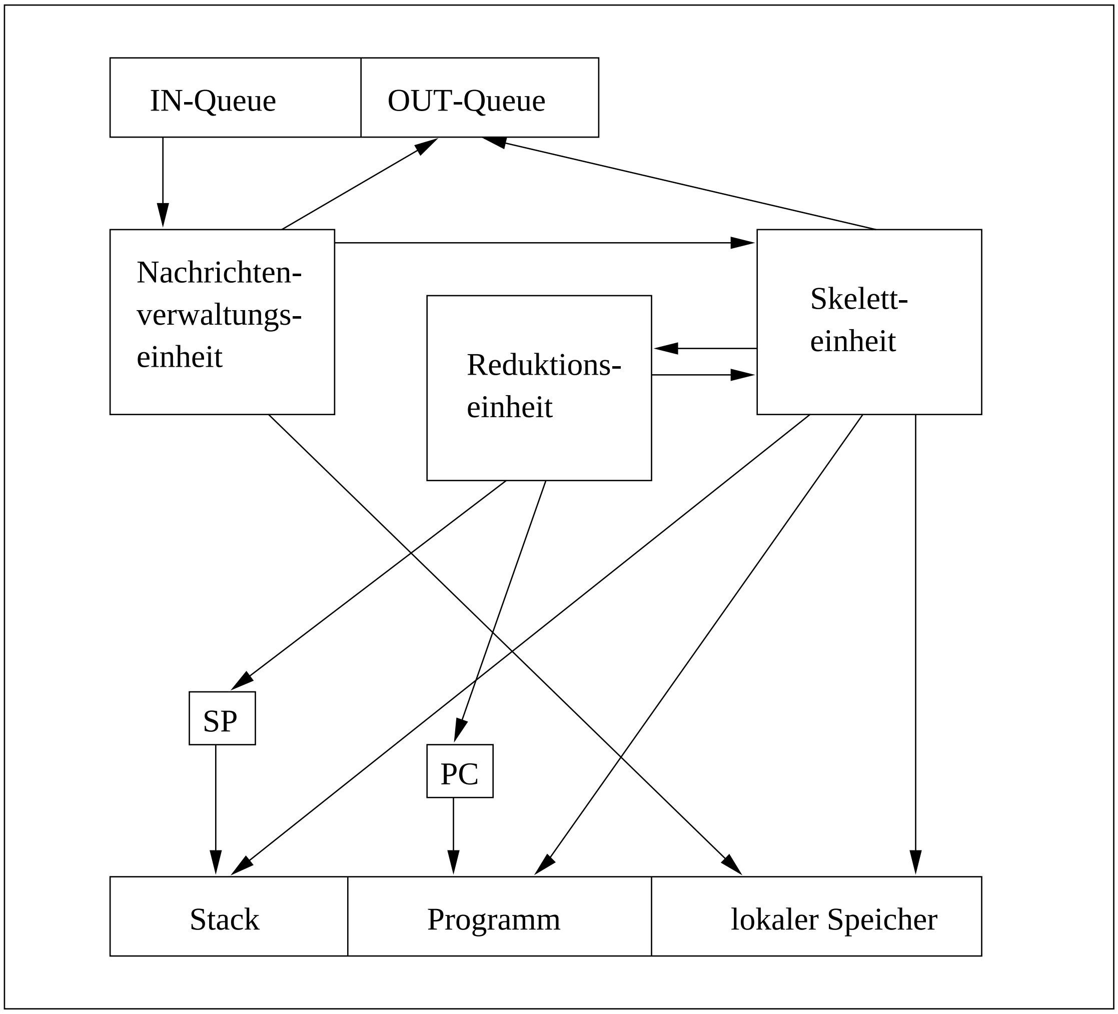


Abbildung 3: Abstrakte Maschine

stellt der PAM-Server dar. Er stellt die Ein- und Ausgabeschnittstelle bereit und übernimmt den sequentiellen Teil der Reduktion des funktionalen Programms. Die Skelette stellen die einzige Quelle der Parallelität in den Programmen dar. Enthält ein Programm keine Skelette, so wird es auf dem PAM-Server sequentiell abgearbeitet. Enthält hingegen ein Programm Skelette, so kommt es zum Skelettaufruf auf dem Server. Die der Reduktionseinheit angeschlossene Skeletteinheit initiiert die Versendung einer Nachricht zur Skelettabarbeitung einschließlich etwa benötigter Daten an alle anderen zu beteiligenden abstrakten Maschinen, die PAM-Worker. Es kommt zur verteilten Abarbeitung des Skeletts mit notwendigen Kommunikationen. Die Teilergebnisse werden mittels Ergebnisnachrichten an den Server gesendet, der das Gesamtergebnis bildet, gegebenenfalls weitere Reduktionen ausführt, weitere Skelettberechnungen initiiert und letztendlich das Resultat der Berechnungen ausgibt.

4.3 Branch-and-Bound als Skelett

Die deklarative Bedeutung von Branch-and-Bound kann wie folgt durch die Definition in einer funktionalen Sprache angegeben werden. Diese Definition enthält zunächst keine Parallelität.

```
VAL branch_and_bound :
[α] → (α → (α → β) → [α]) → (α → β) → ([α] → α) → ([α] → [α]) → γ;

knotenliste branch bound select elimination ⇒
    IF TEST (nil,
         (optimum knotenliste branch bound select elimination))
    THEN branch_and_bound
           (insertation (branch (select knotenliste) bound)
            (elimination knotenliste))
           branch bound select elimination
    ELSE optimum knotenliste branch bound select elimination
    FI;
```

Branch-and-Bound wurde für die Sprache DFS als taskparalleles Skelett in der Sprache C für einen Parallelrechner PARSYTEC MultiCluster 2 mit 64 frei verschaltbaren Transputern des Typs T800 - 20 MHz bzw. einen GigaCluster GCel1024 mit 1024 Transputern T805 - 30 MHz unter dem Betriebssystem Parix implementiert.

Diese Implementierung läßt sich nach Trienekens' Klassifikation (s. Kapitel 3.2) wie folgt einordnen:

Da es sich bei beiden Hardwareplattformen um Systeme mit verteiltem Speicher handelt, bieten sich *lokale, partielle Wissensbasen* an. Der durch einen Zugriff aller Prozesse auf eine gemeinsame Wissensbasis entstehende Flaschenhals wird vermieden, allerdings auf Kosten der Aktualität der Wissensbasis. Jeder Prozeß nutzt jetzt lediglich das Wissen aus seiner lokalen Wissensbasis und legt auch nur dort das von ihm erzeugte neue Wissen ab (*Aktualisierungs-Strategie*). Eine Ausnahme bildet dabei die Berechnung eines lokalen Optimums. Wurde ein solches berechnet, so wird der Wert an alle Prozesse sofort weitergegeben, die, wenn sie eine Teilaufgabe beendet haben, ihre Wissensbasis aktualisieren.

Die *Zugriffs-Strategie* ist dadurch gekennzeichnet, daß jeder Prozeß immer dann auf seine lokale Wissensbasis zugreifen kann, wenn er eine Teilaufgabe beendet hat, die Ergebnisse in der Wissensbasis ablegen möchte und eine neu zu lösende Teilaufgabe auswählen will. Hat ein Prozeß ein lokales Optimum berechnet bzw. es wurde ein solches von einem der anderen Prozesse gesendet, dann vergleicht der Prozeß dieses Optimum mit seinem bisherigen lokalen Optimum und den noch zu lösenden Teilaufgaben. Die *Reaktions-Strategie* besagt, daß ein Prozeß, wenn er ein lokales Optimum von anderen Prozessen erhält, seine derzeitige Arbeit nicht unterbricht, sondern zunächst beendet, und dann erst entsprechend auf die Nachricht reagiert. Es kann dadurch dazu kommen, daß der Prozeß unnötige Arbeit ausführt. Gleichzeitig ist die Größe dieser Arbeit aber aufgrund der Größe der Arbeitseinheiten gering. Außerdem erleichtert diese Form der Reaktions-Strategie die Steuerung der Abläufe.

Eine *Arbeitseinheit* bzw. Teilaufgabe besteht in der Verzweigung eines Problems in Subprobleme und der Berechnung der Schranken für die neu erzeugten Subprobleme. Die *Lastbalancierungs-Strategie* ist sehr einfach. Hat ein Prozessor keine Arbeit mehr, so fragt er reihum seine Nachbarn nach Arbeit. Erhält ein Prozeß eine solche Anfrage, so gibt er eine Arbeitseinheit ab, wobei diese so gewählt wird, daß sie dem der Wurzel am nächsten befindlichen und noch nicht bearbeiteten Knoten im Problembaum entspricht. Die Lastbalancierungs-Strategie ist dynamisch entsprechend der Problemstruktur, denn während der Abarbeitung werden noch neue Subprobleme erzeugt. Sie ist dezentral und damit passend zur Form der Wissensbasis aufgrund der unterliegenden Rechnerarchitektur.

Die Arbeitsweise des implementierten verteilten Algorithmus ist *asynchron*. Das entspricht der Lokalität der Wissensbasen, der Aktualisierungs-, Zugriffs-, Reaktions- und Lastverteilungs-Strategie.

Die Basis-Regeln `branch, bound, select` und `elimination` sind in der Skelettimplementierung selbst nicht berücksichtigt, sondern werden durch den Nutzer in der funktionalen Sprache definiert. Dann kann man das Branch-and-Bound-Skelett auf diese Funktionen zusammen mit dem zu lösenden Problem anwenden.

4.4 Lösung von Maschinenbelegungsproblemen mittels Branch-and-Bound in DFS

Definiert man die *Basis-Regeln* für ein Maschinenbelegungsproblem entsprechend Kapitel 3.3 in der funktionalen Sprache DFS, so kann man das nun zur Verfügung stehende Skelett `branch_and_bound` importieren und ein gegebenes Maschinenbelegungsproblem `init` lösen.

`IMPORT branch_and_bound :`
$[\alpha] \rightarrow (\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow [\alpha]) \rightarrow (\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow \alpha) \rightarrow ([\alpha] \rightarrow [\alpha]) \rightarrow \gamma$;

`EVAL branch_and_bound init branch1 bound1 select1 elimination1 .`

Beispiel 4.1 aus [3]

Vorausgesetzt werden die Basis-Regeln aus Kapitel 3.3. Fünf Produkte $p_1, \dots, p_5$ durchlaufen jeweils zunächst Maschine M_1 und danach Maschine M_2. Die Reihenfolge der

Bearbeitung der Produkte ist auf beiden Maschinen gleich. Die Bearbeitungszeiten $t_{i,j}$, $i = 1, 2$; $j = 1, \ldots, 5$; seien wie folgt gegeben:

Produkt	p_1	p_2	p_3	p_4	p_5
M_1	3	6	2	4	8
M_2	2	5	3	9	3

Gesucht ist die Reihenfolge, die zu minimaler Gesamtzeit bis zur Fertigstellung aller fünf Produkte führt.

Abbildung 4 zeigt einen Teil des sich während der Berechnung aufspannenden Problembaums.

Die Anzahl der möglichen Reihenfolgen ist 5! = 120. Die im Bild ablesbare Optimallösung gehört zum Subproblem $P_{([1,5,2,4,3],5)} = (23, 25)$ mit $U(P_{([1,5,2,4,3],5)}) = 3$. Die Schranke bezeichnet die Stillstandszeit der Maschine M_2. Die minimale Gesamtzeit bis zur Fertigstellung aller Produkte ergibt sich aus der Summe dieser Stillstandszeit $t_{2,s} = 3$ und der Gesamtabarbeitungszeit der Maschine M_2 $\Sigma_{j=1}^{5} t_{2,j} = 25$. Damit kennzeichnet die Stillstandszeit auch die Gesamtarbeitszeit.

Anhand der Auswahlregel `select` wird zuerst immer das Problem mit minimaler Schranke betrachtet und bei mehreren gleichen Schranken das Problem unter diesen Problemen, dessen Knoten sich in maximaler Tiefe im Baum befindet. Bei Gleichheit wird hier unter den Problemen das dem am weitesten links im Baum befindlichen Knoten entsprechende ausgewählt.

Bei sequentiellem Branch-and-Bound werden unter der angegebenen Auswahlvorschrift genau die in Abbildung 4 angegebenen Knoten erzeugt. Bei parallelem Branch-and-Bound kommt es dazu, daß weitere Knoten dekomponiert werden (`branch`), weil Subprobleme, z.B. $P_{([1],1)}, \ldots, P_{([5],1)}$, an parallel arbeitende Prozesse abgegeben werden. Dadurch ergibt sich, daß ein größerer Teil des Baums berechnet wird, allerdings arbeiten hierbei mehrere Prozesse gleichzeitig. Das erweist sich als besonders vorteilhaft, wenn große Teile des Baums berechnet werden müssen.

Die folgende Tabelle zeigt Laufzeitmessungen verschiedener Maschinenbelegungsprobleme. Jedes der Probleme wurde dazu erstens in einer rein funktionalen Weise und zweitens mittels eines Branch-and-Bound-Skeletts beschrieben. Gemessen wurde auf einem Parsytec MultiCluster 2 sowie auf einem GigaCluster GCel1024 und jeweils mit verschiedenen Anzahlen an Prozessoren. Der Parsytec MultiCluster 2 besitzt 64 frei verschaltbare Transputer des Typs T800 - 20 MHz mit je 1 MByte lokalem Speicher, der GigaCluster GCel1024 1024 Transputer T805 - 30 MHz mit je 4 MByte lokalem Speicher.

Es wurde mit verschiedenen Maschinenbelegungsproblemen gemessen. Fall (c) stellt das im obigen Beispiel 4.1 vorgestellte Maschinenbelegungsproblem dar. In Fall (a) muß der gesamte Problembaum aufgespannt, in Fall (b) lediglich ein Zweig des Baums berechnet werden. In allen drei Fällen handelt es sich um Probleme mit 2 Maschinen und 5 Produkten.

Abbildung 4: Teil des Problembaums für das gegebene Maschinenbelegungsproblem

	1 Proz. funktional	1 Proz. Skelett	4 Proz. Skelett	6 Proz. Skelett	8 Proz. Skelett	16 Proz. Skelett
GCel1024 (a)	2.652 s	13.444 s	2.486 s	1.502 s	1.359 s	1.031 s
GCel1024 (b)	0.081 s	0.426 s	0.757 s	0.774 s	0.747 s	0.727 s
GCel1024 (c)	0.111 s	0.586 s	0.888 s	0.854 s	0.852 s	0.797 s
MultiCluster 2 (c)	5.252 s	0.663 s	0.581 s	0.593 s	0.552 s	0.562 s

Die Anzahl der möglichen Reihenfolgen der Maschinenbelegungen ist in allen drei Fällen (a) $5! = 120$. Das bedeutet, daß der gesamte Problembaum 120 Blattknoten besitzt.

Fall (a) beschreibt ein Problem, bei dem es nötig ist, den gesamten Problembaum zu berechnen. D.h., daß insgesamt 326 Knoten (120 Blattknoten + 206 Knoten in Zwischenschichten des Baums) einschließlich der unteren Schranken zu berechnen waren. Man erkennt eine deutliche Abnahme der Laufzeiten aufgrund der Verteilung der Subprobleme auf verschiedene Prozessoren. Vergleicht man die Berechnungszeit des Baums durch das Branch-and-Bound-Skelett mit der Laufzeit des äquivalenten rein funktionalen Programms auf jeweils einem Prozessor, so erkennt man, daß die rein funktionale Berechnung schneller ist als die Skelett-basierte. Das resultiert aus bei der Skelett-basierten Berechnung auftretendem Overhead. Dessen Einfluß relativiert sich aber schon beim Vergleich der Berechnungszeiten der funktionalen Version mit der Skelett-basierten, wenn hier die Gesamtlast auf mehrere Prozessoren aufgeteilt wird. Schon die Verteilung der Arbeit auf vier Prozessoren ist zeitlich günstiger als die rein funktionale Berechnung. Sind also große Teile des Problembaums zu berechnen, fällt die Berechnungszeit einer Skelett-basierten Berechnung günstiger aus als die einer rein funktionalen Berechnung. Die rein funktionale Berechnung des vollständigen Baums auf dem MultiCluster 2 war wegen unzureichenden Speichers, auch nach häufiger Speicherbereinigung, nicht möglich.

Fall (b) zeigt Berechnungszeiten für den entgegengesetzten Extremfall: hier ist im sequentiellen Fall zur Ermittlung eines Optimums lediglich die Berechnung eines Zweiges des Baums notwendig (das entspricht insgesamt 16 Knoten). Die Berechnungszeiten im Fall der Skelettanwendung auf einem Prozessor (0.426 s) ist hier etwas geringer als im Fall der Beteiligung mehrerer Prozessoren an der Berechnung. Das erklärt sich aus der beim Rechnen mit mehreren Prozessoren auftretenden Kommunikation, die bei lediglich einem Prozessor entfällt. Da in Fall (b) parallel wie auch sequentiell die Lösung sehr schnell gefunden werden kann, erzeugt die Beteiligung mehrerer Prozessoren hier lediglich Overhead. Zu erkennen ist außerdem, daß die Berechnungszeit mit immer weiterer Zunahme der Prozessoranzahl nicht steigt. Das wiederum ergibt sich aus der Tatsache, daß die für die Abgabe von Arbeit (Anfordern von Arbeit, Auswahl der zu versendenden Arbeitseinheit, Versenden der Arbeitseinheit) notwendige Zeit im Durchschnitt größer ist als die Zeit zur Dekomposition eines Knotens. Damit wird Arbeit nur dann verteilt, wenn die Verteilung wirklich auch eine Teilung des Arbeitsaufwandes für den sendenden Prozessor bedeutet. Sonst hat der Prozessor bereits alle vorhandene Arbeit erledigt, bevor es überhaupt zur Abgabe von Arbeit kommen kann. Desweiteren ist ersichtlich, daß die rein funktionale Berechnung im Vergleich zur Skelett-basierten schneller ist. Bei der Skelett-basierten Berechnung tritt Overhead durch den expliziten Aufruf der Skeletteinheit, die bei mehreren Prozessoren notwendige Kommunikation und die gesonderte Behandlung der Skelett-Funktionen auf, wozu z.B. das explizite Laden und Entladen des Stacks zählen. Außerdem entfallen in der rein funktionalen Version anwendbare Optimierungen.

Fall (c) zeigt die Laufzeiten des im Beispiel 4.1 dargestellten Maschinenbelegungsproblems; im sequentiellen Fall sind hier 24 Knoten zu berechnen. Im Fall der Berechnung auf dem GCel1024 zeigen sich die gleichen Phänomene wie im Fall (b), wobei hier die Laufzeiten geringfügig größer sind als im Fall (b). Das resultiert daraus, daß hier mehr Knoten des Baums entwickelt werden müssen. Auch im Fall der Berechnung auf dem MultiCluster 2 gelten die Aussagen zu Fall (b). Ausnahme bildet hier allerdings die Laufzeit der rein funktionalen Version des Programms (5.252 s). Ursache dafür ist die hier häufig nötige Speicherbereinigung. Jeder der Prozessoren des MultiClusters 2 hat 1 MByte lokalen Speicher zur Verfügung, im Vergleich zu 4 MByte lokalem Speicher beim GigaCluster GCel1024.

5 Zusammenfassung

Die funktionale experimentelle Sprache DFS wurde um ein Branch-and-Bound-Skelett erweitert. Es wurden verschiedene Maschinenbelegungsprobleme jeweils in einer rein funktionalen Weise und zusätzlich mittels des Branch-and-Bound-Skeletts beschrieben. Damit wurden auf einem Parsytec MultiCluster 2 sowie auf einem GigaCluster GCel1024 und jeweils mit verschiedenen Anzahlen an Prozessoren Berechnungszeitmessungen durchgeführt. Aus diesen Messungen kann man folgern, daß die Verwendung des implementierten Skeletts dann zeitgünstiger ist, wenn ein großer Teil des von dem Maschinenbelegungsproblem aufgespannten Problembaums berechnet werden muß oder wenn aufgrund der großen Anzahl der zu berechnenden Subprobleme in der funktionalen Version häufig eine Speicherbereinigung ausgelöst wird. Weiterhin hat sich gezeigt, daß die Hinzunahme weiterer Prozessoren bei der Berechnung keinen signifikanten Overhead erzeugt.

Weitere Verbesserungen können nun noch durch angepaßte Veränderungen der Strategien des Branch-and-Bound-Algorithmus und eine auf die aktuelle Belastung der Einzelprozessoren abgestimmte Lastbalancierung erreicht werden. Dabei sollte auch die Größe der abzugebenden Arbeitseinheiten von der aktuellen lokalen Last abhängig gemacht werden.

Auch hat sich gezeigt, daß das Konzept der Funktionen höherer Ordnung ein zeitintensives Problem bei der Implementierung funktionaler Sprachen darstellt. Beschleunigungen sind hier durch die Erkennung solcher Funktionen und Umwandlung in Funktionen erster Ordnung, d.h. Instanziierung von Funktionen höherer Ordnung durch Higher-Order-Makro-Technik, falls möglich, zu erwarten. Eine ähnliche Technik kann zur Umwandlung polymorpher Funktionen in monomorphe Funktionen angewendet werden.

Beispiel 5.1 Die Funktion `map` ist eine Funktion höherer Ordnung. Funktionen höherer Ordnung sind Funktionen, deren Argumente oder Resultate selbst wieder Funktionen sind. `map` nimmt als Argumente eine Funktion vom Typ (`INT` $\rightarrow$ `INT`) und eine Liste vom

Typ [INT] und liefert als Ergebnis eine Liste vom Typ [INT], die durch Anwenden der Funktion auf alle Elemente der Argumentliste entsteht.

```
VAL map : (INT → INT) → [INT] → [INT];
f liste ⇒
 IF TEST (nil, liste) THEN nil
 ELSE cons (f SELECT(cons, 0, liste)) (map f SELECT(cons,1, liste))
 FI;
```

Außerdem ist die Funktion plus_zwei gegeben, die zu einer Integer-Zahl 2 hinzuaddiert:

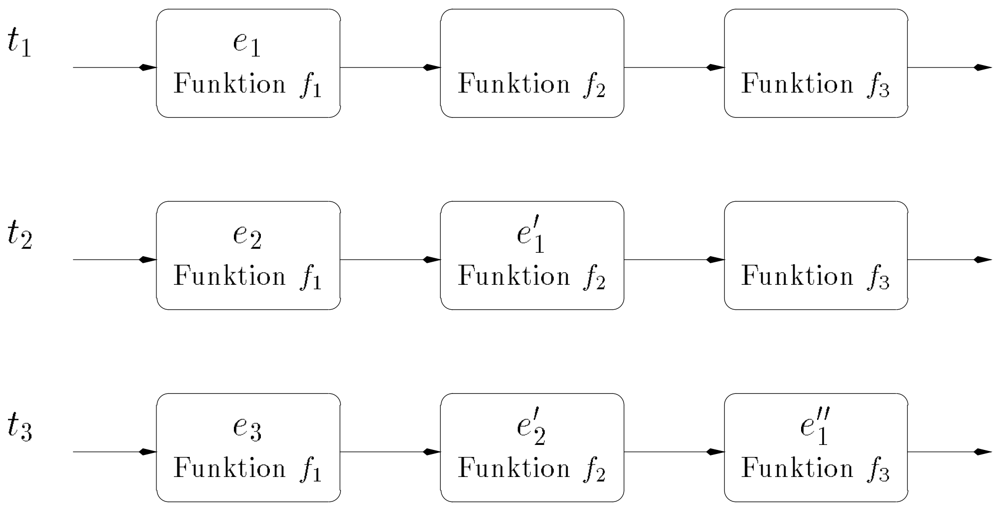
```
VAL plus_zwei : INT → INT;
x ⇒ x+2;
```

Wird die Funktion map mit der Funktion plus_zwei instanziiert, so erhält man eine Funktion erster Ordnung map_1:

```
VAL map_1 : [INT] → [INT];
liste ⇒
 IF TEST (nil, liste) THEN nil
 ELSE cons (plus_zwei SELECT(cons, 0, liste))
           (map_1 SELECT(cons,1, liste))
 FI;
```

Darüber hinaus hat sich bei der Untersuchung weiterer Verfahren in bezug auf deren Skelettierbarkeit gezeigt, daß das Hindley-Milner-Typ-System nicht mehr in allen Fällen ausreicht. Zur Einführung allgemeiner Skelette wie farm oder pipe sind existentielle Typen notwendig.

Beispiel 5.2 Ein Pipeline-System oder eine Pipe besteht hierbei aus einer Folge von Funktionen polymorphen Typs $(\alpha_{i-1} \rightarrow \alpha_i),\ i = 1, 2, \ldots,$ die jeweils nacheinander auf jeweils ein Element einer Argumentliste angewendet werden. Dabei ist das Resultat vom Typ α_i einer Funktion vom Typ $(\alpha_{i-1} \rightarrow \alpha_i)$ das Argument der Folgefunktion vom Typ $(\alpha_i \rightarrow \alpha_{i+1}),\ i = 1, 2, \ldots$ Die Elemente der Argumentliste $[e_1, e_2, e_3, \ldots]$ durchlaufen nacheinander die Funktionen, so daß z.B., wenn ein Element e_1 zum Zeitpunkt t_1 die Funktion f_1 passiert, das Ergebnis e_1' der Anwendung von f_1 auf e_1 zum nachfolgenden Zeitpunkt t_2 die Funktion f_2 und ein Element e_2 zu diesem Zeitpunkt die Funktion f_1 passieren kann. Zum darauffolgenden Zeitpunkt t_3 wird dann auf e_1'' die Funktion f_3, auf e_2' die Funktion f_2 und auf e_3 die Funktion f_1 angewendet:



Die Import-Deklaration für ein pipe-Skelett hätte dann folgendes Aussehen:

`IMPORT pipe :`
$((\alpha_0 \to \alpha_1) \to (\alpha_1 \to \alpha_2) \to \ldots \to (\alpha_{n-1} \to \alpha_n)) \to [\alpha_0] \to [\alpha_n]$

Allerdings ist die Anzahl der zu durchlaufenden Funktionen beliebig und nicht von vornherein bekannt, so daß man den Typ der Skelett-Funktion nicht in dieser Weise angeben kann. Eine Möglichkeit wäre hier eine Typdefinition in ähnlicher Weise wie die Typdefinition von Listen von Elementen gleichen Typs:

`TYPE list` α `= nil` $|$ `cons OF` $\alpha \to$ `list` α

`TYPE fkt` $\alpha\ \gamma$ `= fnil` $|$ `fcons OF` $(\alpha \to \beta) \to$ `(fkt` $\beta\ \gamma$`)`

Hierbei ist jetzt allerdings die Verwendung eines existentiellen Typs β nötig, was im Hindley-Milner-Typ-System nicht möglich ist.

Bei Einführung existentieller Typen läßt sich dann unter Angabe obiger Typdefinition ein pipe-Skelett mit korrekter Typangabe definieren und verwenden.

`IMPORT pipe :`
`fkt` $\alpha_0\ \alpha_n \to [\alpha_0] \to [\alpha_n]$

Neben der direkten Bereitstellung der Skelette für den Anwender scheint eine Nutzung dieser in deklarativen Sprachen zur Erweiterung der Sprachen um Constraint-Programming-Systeme sinnvoll. So können z.B. für Constraint-Solver für lineare Constraints über rationalen Zahlen zur Implementierung des Simplexalgorithmus datenparallele Skelette über Arrays genutzt werden und für Constraint-Solver für Constraints über endlichen Mengen von Variablen mit zugeordnetem endlichen Wertebereich das Branch-and-Bound-Skelett.

Literatur

[1] Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In: Proceedings of EuroPar '96, Vol.1, Lecture Notes in Computer Science 1123. Berlin 1996. S.718-731.

[2] Farreira, A., Pardalos, P.(Eds.): Solving Combinatorial Optimization Problems in Parallel. Lecture Notes in Computer Science 1054. Berlin 1996.

[3] Grosche,G., Ziegler, V. (Hrsg.): Ergänzende Kapitel zu: Bronstein, I.N., Semendjajew K.A.: Taschenbuch der Mathematik. Leipzig 1981.

[4] Kovács, L.B.: Combinatorial Methods of Discrete Programming. Budapest 1980.

[5] Kuchen, H.: Skeletons versus C - A Case Study. International Workshop on the Implementation of Functional Languages, Baastad, Sweden, 1995. Proceedings appeared as Report of the Chalmers University of Technology, Göteborg. 1995.

[6] Mattern, F.: Verteilte Basisalgorithmen. Informatik-Fachberichte 226. Berlin 1989.

[7] Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization. Englewood Cliffs, NJ 1982.

[8] Park, S.-B.: Implementierung einer datenparallelen funktionalen Programmiersprache auf einem Transputersystem. Diplomarbeit. RWTH Aachen 1995.

[9] Stoltze, H.: Implementierung einer parallelen funktionalen Sprache mit algorithmischen Skeletten zur Lösung mathematisch-technischer Probleme. Dissertation. RWTH Aachen 1994.

[10] Trienekens, H.W.J.M.: Parallel Branch and Bound Algorithms. Dissertation. Universität Rotterdam 1990.

[11] Wagener, A.: Lastverteilungen und Netzwerktopologien für eine Realisierung der parallelen abstrakten Maschine PAM. Diplomarbeit. RWTH Aachen 1990.

Anhang

<programm> ::= {<typedef>}{<combinatordef>} EVAL <expr> .
<typdef> ::= TYPE <typeident> = <alternative> { | <alternative>};
<typeident> ::= <identifier>
<alternative> ::= <constructor> [OF <carttype>]
<constructor> ::= <identifier>
<type> ::= <carttype> [→ <type>]
<carttype> ::= <basetype> [* <carttype>]
<basetype> ::= INT | BOOL | CHAR | REAL | ARRAY | (<type>) | <typeident>()
<combinatordef> ::= VAL <combinator>: <type>; {<variable>} => <expr>; |
IMPORT <combinator>: <type>;
<combinator> ::= <identifier>
<variable> ::= <identifier> | <pattern>
<pattern> ::= (<identifier>, <variable>)
<explist> ::= <exp> {<exp>}
<exp> ::= (<explist> {,<explist>}) | <constant> | <variable> |
<combinator> | <constructor> |
<explist> <operation> <explist> |
EVEN <explist> | ODD <explist> | NOT <explist> |
SELECT (<constructor>,<intnumber>,<explist>) |
TEST (<constructor>,<explist>) |
IF <explist> THEN <explist> ELSE <explist> FI |
LET <variable> = <explist> {;<variable> = <explist>}
IN <explist> END
<constant> ::= <intnumber> | <realnumber> | <character> | TRUE | FALSE
<operation> ::= AND | OR | + | - | * | / | MOD | = | != | < | > | ≤ | ≥ |
XOR | >> | <<

<identifier> besteht aus einer mit einem Buchstaben beginnenden Folge von Buchstaben und Ziffern. <intnumber> setzt sich aus einer Folge von Ziffern zusammen, <realnumber> aus einer Folge von Ziffern, wobei ein Punkt und/oder ein durch 'e' bzw. 'E' abgetrennter Exponent zusätzlich in üblicher Weise zulässig sind. <charakter> bezeichnet ein druckbares Zeichen.