Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20) Praktische Informatik *VSFT*

Übersicht

über das Parallelisierungskonzept von ATTEMPTO 2

R.Brause

In der Nachfolgeversion von ATTEMPTO wurde ein Schema für eine Parallelarbeit mit feinerer Granularität, aber auch komplizierterem Ablauf konzipiert [LUT1], [LUT2] und implementiert [KAM], über das hier ein kurzer Überblick gegeben werden soll.

Ausgehend von der Überlegung, daß es sinnvoller ist, die dem Problem inhärente Parallelität *explizit* im Programm auszudrücken als dies einem "intelligentem"(?) Compiler zu überlassen, wählten wir als Parallelisierungsebene die *Spezifikation* im Programm durch das PARBEGIN/PAREND Konstrukt. Die Implementierung in dem nachrichtenorientierten ATTEMPTO1-System durch *send()* und *receive()* Aufrufe sollte dabei dem Programmierer verborgen bleiben.

Dies läßt sich in folgenden Konzept-Punkten beschreiben (s. [LUT1]):

- Der Programmierer spezifiziert in seinem Programm nur die Menge der parallel ausführbaren Aktionen; die Konfiguration und Mechanismus der Verteilung auf die vorhandenen Prozessoren bleiben verborgen.
- Das Programm soll unabhängig von der Zahl der bei der Ausführung beteiligten Prozessoren das selbe Ergebnis bringen. Damit sind Umkonfigurierung des Systems (Ausfälle!) und Portierung des Programms leicht möglich.
- Während der Compilierung sollte eine möglichst weitgehende, maschinelle Überprüfung der konsistenten Nutzung von parallelen Konstrukten im Programm durchgeführt und die nachrichtenorientierte Implementierung durch *send()* und *receive()* Aufrufe automatisch generiert werden.
- Der Aufruf und die Parameterübergabe von parallel ablaufenden Aktionen sollte automatisch generiert werden. Auch die Eingliederung erhaltener Ergebnisse erfolgt automatisch.

Dazu sollen folgende Sprachkonventionen verwendet werden:

- *PARBEGIN und PAREND*
  Alle parallel ausführbaren Aktionen sind zwischen PARBEGIN und PAREND aufgeführt. Damit ist PAREND als Barriere der Punkt, an dem die parallelen Aktionen terminieren und die Ergebnisse der Aktionen verarbeitet werden müssen.
- *Mittlere Granularität*
  Die mit PARBEGIN und PAREND geklammerten, parallel ausführbaren Aktionen (*paralleler Bereich*) sind keine Elementarbefehle (feine Granularität), sondern ganze Befehlsabschnitte (mittlere Granularität). Dies ist besonders bei nachrichtenorientierten Systemen sinnvoll, da der Mehraufwand für Aufruf und Parameterübergabe von parallel ablaufenden Aktionen meist mehrere Elementarbefehle ausmacht.
  Der sequentielle Befehlsabschnitt kann mit extra Klammern (z.B. *PB* und *PE*) gekennzeichnet werden, oder aber durch die *Einschränkung der parallelen Aktionen auf Prozeduren.*
- *Makro-Anweisungen*
  Werden viele, gleichartige, aber mit unterschiedlichen Parametern aufgerufene Aktionen in einem parallelen Bereich verwendet, so läßt sich dies mit *Makro-Anweisungen* (z.B. FOR i:=1 TO 10 DO) spezifizieren, die dann zur Compile-Zeit zu Anweisungen für echt parallele Aktionen expandiert werden.

Die Einführung eines *pipe-Konstrukts* (z.B. *PIPEBEGIN / PIPEEND*) für pipeline-ähnliche Parallelität kann man zunächst verzichten, da bereits mit PARBEGIN/PAREND eine pipeline

installiert werden kann. In der folgenden Abbildung ist ein Beispiel einer zweistufigen Pipeline gegeben, bei dem das Hauptprogramm die Rolle der Arbeitsvermittlung (Übertragung) zwischen der Prozedur p1 und der Prozedur p2 einnimmt.

```
Y:=NULL;
LOOP
Input(X);                          (* Neues X *)
  PARBEGIN
        p1 (VAR X);                (* als Eingabe der ersten Stufe *)
        p2 (VAR Y);
  PAREND                           (* Alter Wert von X wird überschrieben *)
Output(Y); Y:=X;                   (* und zur Eingabe Y der nächsten Stufe *)
ENDLOOP
```

Abb. 1 Pipeline-Konstruktion mit PARBEGIN/PAREND

Das Implementierungskonzept

Die Umsetzung der Spezifikation in tatsächlich parallel ablaufende Aktionen läßt sich auf verschieden Wegen erreichen. Der vollständigste Ansätz dürfte sicher in der Erstellung eines eigenständigen, neuen Compilers einer neuen, parallelen Sprache liegen. Dem stehen allerdings Probleme wie Portabilität, Arbeitsaufwand und Stabilität einer Neuimplementierung gegenüber.

Stattdessen entschieden wir uns für die Idee, die Prozeduren eines parallelen Bereichs als "remote procedure calls" (RPC) aufzurufen. Ordnet man jedem RPC einen Leichtgewichtsprozeß zu, so realisieren die unabhängig voneinander blockierenden Leichtgewichtsprozesse parallel ablaufende "remote service invocations" (RSI) des Hauptprogramms [KAM].

Dieses Konzept ermöglichte uns, anstelle einer neuen Programmiersprache eine bestehende, blockorientierte Sprache (Modula-2) mit parallelen Sprachkonstrukten dadurch zu erweitern, daß man die Schlüsselworte von einem Präprozessor im Programmtext durch Laufzeitprozeduren einer besonderen Bibliothek ersetzen läßt.
Dies bedeutet insbesondere (s. [LUT2]):

- PARBEGIN und PAREND werden in Laufzeitprozeduren ParBegin(..) und ParEnd(..) umgesetzt. Diese Prozeduren aktualisieren die interne Nummer des parallelen Bereichs und stoßen eine Prozedur an, die auf alle Ergebnisse wartet und sie dann eingliedert.
- Für jede in einem parallelen Bereich aufgerufene Prozedur generiert der Präprozessor eine Stub-Prozedur (s. Abschnitt 1.3.3), die die Zwischenschicht zwischen Prozeduraufruf und Nachrichtenaustausch realisiert.
- Beim Aufruf einer Prozedur im parallelen Bereich wird bei der Übergabe eines Referenzparameters (VAR-Parameter in Modula-2) nicht nur die Parameteradresse, sondern auch der Parameterwert sowie seine Speichergröße (SIZE) übergeben. Adresse und Größe werden von ParEnd() ausgewertet.
- Kopien des Hauptprogramms mit allen Prozeduren werden auf den beteiligten Prozessoren gehalten. Die Unterscheidung in Hauptprogramm oder *Auftraggeber* (AG) und aufgerufene Prozedur-Service oder *Auftragnehmer* (AN) wird nach der Prozeßverteilung bei der Initialisierung vorgenommen.
- In den RPC-Prozeduren dürfen keine globalen Variablen verwendet werden.

Die Überprüfung des Gebrauchs globaler Variable läßt sich modularisieren, wobei die Module um Pseudokommentare erweitert werden [LUT2]. Der falsche Gebrauch von globalen Variablen läßt sich allerdings nicht völlig vermeiden; hier müßte sonst der Compiler mit allen verfügbaren Informationen umgeschrieben und erweitert werden.

Ist das Zielsystem ein homogenes, aus gleichartigen Prozessoren bestehendes Multi-Mikroprozessorsystem wie ATTEMPTO 1, so kann der RPC sehr schnell und effektiv durch das Verschicken des Prozeduraufruf-Stacks in den Auftragsnachrichten implementiert werden [LUT2].

Die Lastverteilung der RSI auf die beteiligten Prozessoren sollte sowohl vom Kommunikationsaufwand (Zahl und Größe der Parameter, etc) als auch von der Laufzeit-Länge der parallelen Aktionen abhängen. Um für einen guten Schedule Anhaltspunkte zu gewinnen, wird das Programm zunächst auf einem einzigen Prozessor gestartet und ein Ausführungsprofil erstellt. Mit Hilfe dieser ersten Schätzung und den Informationen über die parallele Struktur, die der Präprozessor erstellt hat, kann nun ein Scheduler eine Tabelle erstellen, die zur Laufzeit eine statische, im Mittelwert optimale Verteilung bewirkt. Dabei kann dieser Schedul nur eine suboptimale Näherung sein, da das Problem NP-vollständig ist. Eine geeignete, suboptimale Scheduling-Strategie zu finden ist allerdings noch Gegenstand der Forschung.

Literatur

[KAM] K.Kammers
Parallelisierung prozeduraler, blockorientierter Programmiersprachen
Diplomarbeit am Fachbereich Informatik
der J.W.-Goethe Universität, Frankfurt a.M. 1989

[LUT1] J.Lutz, R.Brause, M.DalCin, Th.Philipp
Ein Parallelisierungskonzept für ATTEMPTO 2
Interner Bericht 1/89 des Fachbereichs Informatik
der J.W. Goethe Universität Frankfurt a.M., 1989

[LUT2] J.Lutz, R.Brause, M.DalCin, K.Mamers, Th.Philipp
Die Erweiterungen der ATTEMPTO 2 Laufzeitbibliothek
Interner Bericht 2/89 des Fachbereichs Informatik
der J.W. Goethe Universität Frankfurt a.M., 1989