Ausarbeitung zum Transputer-Praktikum
Sonntag, 07. Mai 1995

Kay Römer
Thomas Holubar
Atilla Ünlü
Frank Pilhofer

Integralberechnung

Aufgabe 1A

Aufgabenstellung

In dieser Aufgabe soll ein bestimmtes Integral einer fest definierten Funktion berechnet werden. Die sequentielle Lösung des Problems unterteilt den Bereich, über den der Wert des Integrals zu bestimmen ist, in eine Anzahl von kleinen Intervallen, in denen das Integral näherungsweise über den Flächeninhalt eines in die Funktion hereingelegtes Rechtecks bestimmt wird. Die Fehler, die dabei entstehen, werden bei Erhöhung der Intervallanzahl immer kleiner (bei unendlich vielen Integralen verschwinden sie -- so ist schließlich das Integral definiert). Um einen möglichst genauen Wert zu erhalten, müssen also so viel Intervalle wie möglich berechnet werden.

Ansatz

Das Schöne an diesem Problem ist, daß die Aufgabe beliebig parallelisierbar ist. Der Näherungswert des Gesamtintegrals ist einfach die Summe der Näherungswerte der Teilintervalle. Prinzipiell könnten alle Einzelergebnisse getrennt berechnet und hinterher addiert werden.

Zur Lösung des Integrals wird der Gesamtbereich zunächst in so viele Teile zerteilt, wie Prozessoren im System vorhanden sind. Diese Teile werden dann innerhalb eines jeden Transputers weiter zerlegt. Auf jedem Prozessor entsteht so eine gute Annäherung an das Integral des jeweiligen Teilbereiches. Sobald alle Ergebnisse feststehen, können sie problemlos zur Lösung des Gesamtproblems addiert werden.

Implementierung

Das Programm wird nach dem Master-Slave-Prinzip aufgebaut. Ein Masterprozeß erhält die Aufgabenstellung, teilt sie in entsprechend viele Teile auf, verteilt die einzelnen Aufgaben auf verschiedene Prozessoren und sammelt schließlich alle Ergebnisse wieder ein. Slaveprozesse

haben nichts weiter zu tun, als die ihnen zugeteilte Aufgabe anzunehmen, die Lösung zu bestimmen. Es wurde versucht, die Slaveprozesse möglichst "dumm" zu gestalten, und die Verwaltungsaufgaben völlig auf den Master zu beschränken.

Die Prozessoren werden in einem Ring organisiert, was mit der folgenden CDL-Datei erreicht wird:

```
master.z <> (| [$1] slave.z)
```

Eine beliebige Anzahl von Slaveprozessen wird hintereinandergeschaltet. Der Master kann vorne in diese Kette hereinschreiben, und am anderen Ende aus ihr lesen. Zwar bedeutet dies, daß Daten immer über mehrere Prozessoren hinweggereicht werden müssen, doch resultiert diese Struktur in einer einfachen und flexiblen Implementation, da insbesondere im Master die Kanalnummern bei allen Konfigurationen unverändert bleiben.

Um das Kommunikationsprotokoll möglichst einfach zu gestalten, findet alle Kommunikation durch Austausch von Paketen fixer Größe statt. Dieses Paket ist eine Struktur, die Platz für alle benötigten Daten enthält. Die Struktur ist wie folgt definiert:

```
#define COUNT         0
#define AUFGABE       1
#define ERGEBNIS      2
#define GO            3
#define QUIT          4

typedef struct {
    int process;
    int command;
    union {
          struct aufgabe {
                double anf, end;
                int schritt;
          } aufg;
          int count;
          double ergebnis;
    } u;
} Comm;
```

Process ist der Prozeß, für den dieses Paket bestimmt ist. *Command* ist eines der mit den #defines definierten Kommandos, die Prozeß erhalten kann. Die union *u* enthält Daten, die zu den verschiedenen Befehlen gehören, wie z.B. die Aufgabenstellung zur Berechnung, oder ein Platzhalter für das zurückzugebende Ergebnis.

Die verschiedenen Befehle haben die folgende Bedeutung:

COUNT Wird bei der Initialisierung des Programmes verwendet, um die Slaveprozesse im System zu zählen. Das Paket wird im Kreis durch alle Prozesse durchgereicht. Jeder Prozeß liest das Feld `u.count`, speichert es als eigene Prozeßnummer ab und erhöht das Feld. Auf diese Weise erhält jeder Prozeß eine eindeutige

Nummer. Das Zählen endet wenn der Master das Paket vom letzten Prozeß erneut einliest.

AUFGABE Der Prozeß erhält die Aufgabe, das Integral über das Teilintervall von `u.aufg.anf` bis `u.aufg.end` in `u.aufg.schritt` Schritten zu berechnen. Der Prozeß merkt sich diese Werte, startet allerdings die Berechnung noch nicht. Denn wenn ein Prozeß in der Kette rechnet, könnten keine Aufgaben mehr an dahinterliegenden Prozesse verschickt werden.

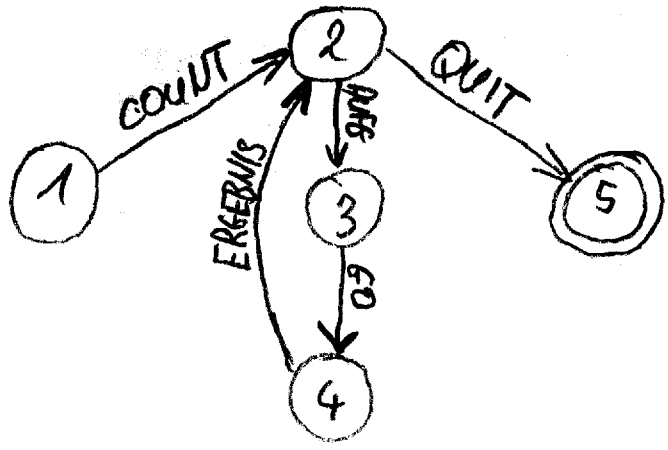
GO Dies ist ein ungerichteter *broadcast*-Befehl (als `process` wird 0 eingetragen). Jeder Prozeß schickt das Paket zuerst weiter, dann wird die Berechnung der zuvor erhaltenen Aufgabe gestartet. Das Ergebnis dieses Teilintegrals wird zunächst gespeichert.

ERGEBNIS Das zuvor berechnete Ergebnis wird gelesen. Der Prozeß, der diesen Befehl erhält, legt sein Ergebnis in `u.ergebnis` ab, und schickt das Paket weiter in der Kette bis zum Master.

QUIT Auch hierbei handelt es sich um einen *broadcast*-Befehl. Er veranlaßt, daß sich die Slaveprozesse beenden. Dabei wird natürlich gleichzeitig die Kommunikationskette zerstört, so daß nach Rundlauf dieses Befehles keine weiteren Aufgaben mehr berechnet werden können.

Der Slaveprozeß

Der Slaveprozeß kann prinzipiell als deterministischer Automat dargestellt werden:



Im Programmtext sind allerdings die Zustände 2 bis 4 zusammengefaßt und nicht voneinander zu unterscheiden. Es bleibt dem Master überlassen, die Befehle immer nur in einer sinnvollen Reihenfolge zu schicken. In C befindet sich der Automat in einer endlosen while-Schleife, die genau dann terminiert, wenn der QUIT-Befehl gelesen wird. In der Schleife werden kontinuierlich Pakete gelesen. Wenn das Paket von Interesse ist (ein Broadcast oder an diesen

Prozeß gerichtet), so wird der Befehl im Paket ausgewertet und ausgeführt. In diesem und in allen anderen Fällen wird das Paket auch noch weiter in die Kette geschrieben. Das ist zwar nicht bei allen Befehlen unbedingt nötig (z.B. bei AUFGABE), doch hat dieser zusätzliche Datenfluß kaum Einfluß auf die Gesamtgeschwindigkeit.

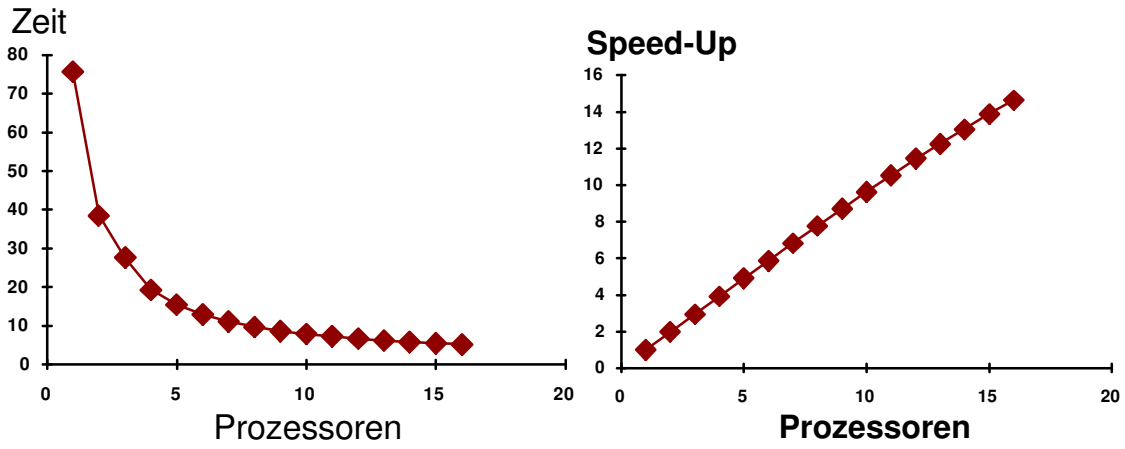
Der Masterprozeß

Der Master muß zunächst die Aufgabe entgegennehmen, sei es aus entsprechenden Programmparametern und von der Tastatur. Anschließend muß die Vielzahl von obigen Zustandsautomaten gesteuert werden. Als erstes zählt der Master die Anzahl von Prozessen im System mit dem COUNT-Befehl. Dann wird die Aufgabe in ebenso viele Teile unterteilt, und als AUFGABE an die einzelnen Prozesse geschickt. Nachdem alle Prozesse bedient sind, wird die Berechnung global mit einem Broadcast vom Typ GO gestartet. Nun wartet der Master auf die Beendigung der Berechnung und liest anschließend von den einzelnen Prozessen mit ERGEBNIS das Teilergebnis ein. Diese werden dann zusammenaddiert und gedruckt. Zum Schluß schickt der Master den QUIT-Befehl und beendet sich selber. Auf diese Weise wird alles sauber beendet.

Performance

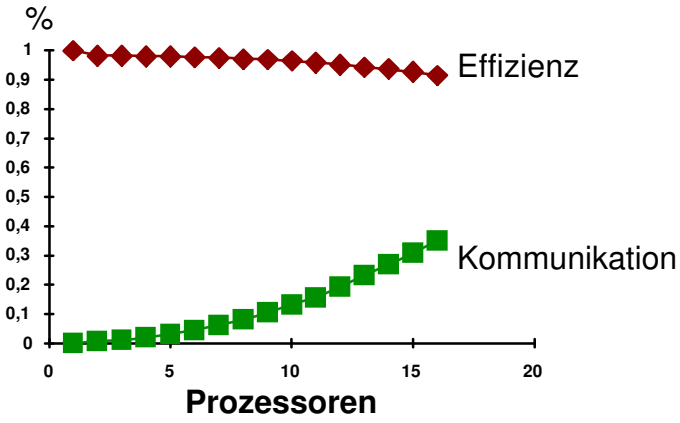
Um den Master wurden nun verschiedene Anzahlen von Slaves konfiguriert und die Laufzeit immer der gleichen Aufgabe (Integral des sinus im Bereich von 0 bis π bei einer Million Intervallen) gemessen. Dabei ergibt sich das folgende Resultat:

Prozessoren	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Zeit (s)	75,8	38,5	25,7	19,3	15,5	12,9	11,1	9,8	8,7	7,8	7,2	6,6	6,2	5,8	5,5	5,2
Speed-Up	1	1,97	2,95	3,92	4,90	5,86	6,83	7,77	8,72	9,64	10,5	11,4	12,3	13,1	13,9	14,6
Effizienz	1	0,99	0,98	0,98	0,98	0,98	0,98	0,97	0,97	0,96	0,95	0,95	0,95	0,94	0,93	0,91
Kommunikation (s)	0	0,28	0,28	0,39	0,49	0,58	0,70	0,79	0,91	1,04	1,12	1,28	1,44	1,56	1,68	1,82



Auf der x-Achse ist die Anzahl der verwendeten Prozessoren dargestellt. Die linke Tabelle

zeigt die verbrauchte Zeit in Sekunden, die rechte Tabelle zeigt den Speedup. Beide Kurven sehen ziemlich ideal aus: Die Zeit ist antiproportional, und der Speedup ist proportional zur Anzahl der Prozessoren. Ein wenig genauer zeigt sich alles im folgenden Diagramm:



Die obere Kurve zeigt die Effizienz, die untere das Verhältnis von Kommunikationszeit zur Rechenzeit. Die Effizienz liegt bei stetigem Abfall nur knapp unter 100%, wenn sich dieser Abfall auch bei mehr Prozessoren leicht beschleunigt, während die relative Kommunikationszeit stärker als linear wächst. Der leichte Effizienzabfall kann auch durch eine kleine Unschärfe erklärt werden; aus technischen Gründen fließt noch etwas Kommunikationszeit in die Messung der Rechenzeit ein.

Diskussion

Die Grafiken zeigen, was zu erwarten war: Das Integralproblem ist sehr gut parallelisierbar. Egal, wieviel Prozessoren eingesetzt werden, es wird fast das Optimum an Geschwindigkeit erreicht. Je mehr Prozessoren allerdings bei der Lösung des Problems mithelfen, desto mehr Daten müssen durch die Gegend geschaufelt werden. Insbesondere wächst dank der Ringstruktur die Anzahl von Datenpaketen quadratisch mit der Anzahl der Prozesse (der genaue Wert ist $P = 3(n+2)+2n(n+2)$). Es wird daher eine Grenze geben, oberhalb der weitere Parallelisierung nicht mehr sinnvoll ist. Mit über 30% Kommunikationszeit bei 16 Prozessoren ist diese Grenze bereits fast erreicht.

Die bestehenden Programme für Master und Slave lassen sich in Hinsicht auf Kommunikation sicher noch verbessern. Bislang wird für jede Aufgabenstellung und jedes Ergebnis eines jeden Prozessors ein komplettes Paket durch den gesamten Ring geschickt. Anders programmiert würde die Paketanzahl nur linear anwachsen. Auch könnte versucht werden, die zugrundeliegende Hardwarearchitektur besser auszunutzen. Alle Testläufe dieses Programms fanden auf der Architektur 4x4 statt; die Zuordnung Prozess zu Prozessor wurde vollständig Helios überlassen.

Solche Methoden würden die sinnvolle Parallelisierungsgrenze weiter hinauszögern, doch früher oder später wird sie immer erreicht werden. Wir haben uns auch bewußt gegen die Ausnutzung der Hardwarearchitektur ausgesprochen, da das Programm mit solchen Mitteln notwendigerweise hardwarespezifisch werden muß und seine Flexibilität einbüßen würde.

Anhang

Im Anhang befinden sich die Listings von:

- master.c
- slave.c
- makefile

Zu dem Projekt gehören auch noch die Dateien integral.cdl und die Strukturdefinition in comm.h. Beide Dateien sind bereits oben vollständig abgedruckt worden.