Ausarbeitung zum Transputer-Praktikum
Sonntag, 02. Juli 1995

Kay Römer
Thomas Holubar
Atilla Ünlü
Frank Pilhofer

Fakultäten

Einleitung

Eines Tages im Jahre 1993 rief mich ein Freund ganz verzweifelt an. Er erzählte von einer Wette, in der er behauptet hatte, es wäre möglich, die Fakultät von 10000 zu berechnen. Er hätte sich auch bereits ein Programm geschrieben, welches den Wert berechnen sollte, doch würde dieses schon seit 24 Stunden laufen und sei erst bei einem Fünftel des Zielwertes angelangt. Diese lange Rechenzeit kam mir unverständlich vor. Direkt nach Ende des Telefonats setzte ich mich auf den Hintern, und bereits am folgenden Morgen warf ich eine Diskette mit der fertig ausgerechneten Fakultät von 10000 in den Briefkasten. Rechenzeit: 1 Stunde.

Nach seinem verblüfften Rückanruf auf den Erhalt der Diskette hin ließ er nicht locker, bevor ich nicht das Prinzip der Berechnung verraten hatte. Daraufhin, ausgehend vom gleichen Standpunkt, begann ein regelrechter Krieg, in dem wir unsere Rechenzeiten ständig unterboten. Aus Stunden wurden Viertelstunden, aus Viertelstunden wurden Minuten. Doch näherten wir uns bald einer optimalen Strategie an, deren Rechenzeiten nicht mehr zu unterbieten waren. Das Programm wurde daraufhin eingesetzt, um möglichst große dieser absolut sinnlosen Fakultäten zu berechnen. Die Fakultät von 200000, eine Zahl von 970000 Ziffern, wurde in knapp 3 Tagen berechnet. Die absolute Schallgrenze stellte die Fakultät von 500000 dar, deren 2.6 Millionen Zeilen glatte 45 Seiten Papier, in einer 3pt-Schrift bedruckt, füllen. Rechenzeit auf einer HP Workstation: 15 Tage. Zu diesem Zeitpunkt entstand auch die Idee, die Rechenzeit durch Parallelisierung weiter zu beschleunigen.

Die Idee wurde nie implementiert und geriet fast zwei Jahre in Vergessenheit, bis sie im Rahmen des Transputerpraktikums wieder ausgegraben wurde. Endlich stand eine Plattform bereit, die die Implementation eines parallelen Algorithmus begünstigte.

Der Datentyp bigint

Klar ist, daß Zahlen wie die Fakultät von 10000 den Zahlenbereich jedes Standarddatentyps sprengen. Bereits die Fakultät von 13 ist größer als `ULONG_MAX`. Deshalb wird ein eigener Datentyp geschaffen, der die nötigen Zahlenwerte halten kann. Die Deklaration des Typs ist wie folgt:

```
typedef struct {
  long length;
  long first;
  long sign;
  unsigned short digits[BIGLENGTH];
} bigint;
```

Die Zahl wird in einem Array zur Basis 10000 gespeichert, d.h. jedes Element des Feldes enthält eine Zahl zwischen 0 und 9999. Die komplette Zahl ergibt sich als $Z = \sum_{i=0}^{BIGLENGTH-1} 10^{4 \cdot i} \cdot \text{digits}[i]$. Zur Optimierung zeigt `first` auf das erste und `length` auf das letzte Element der Zahl ungleich 0 (d.h. der tatsächlich verwendete Bereich der Zahl). Alle Elemente des Arrays außerhalb dieses Bereiches sind 0. Mathematisch formuliert müssen bei der Behandlung einer Zahl die folgenden Bedingungen eingehalten werden:

- $digits[first] > 0 \wedge \forall (i < first): digits[i] = 0$
- $digits[length] > 0 \wedge \forall (i > length): digits[i] = 0$

Eine Ausnahme der beiden obigen Regeln stellt die Zahl 0 dar, bei der alle Elemente des Arrays 0 sind, und `first`, `length` ebenfalls 0 sind. Die beiden Regeln helfen, die Operationen auf den Zahlen zu beschleunigen. Beim Addieren zweier Zahlen beispielsweise brauchen ja nur die Teile der Zahlen addiert zu werden, die ungleich 0 sind.

Mittlerweile steht für den Zahlentyp eine komplette Funktionsbibliothek in C++ zur Verfügung, mit der sich mit `bigints` fast genauso gut wie mit normalen `ints` rechnen läßt. Auf die Beschreibung der Implementationen soll hier verzichtet werden. Im Rahmen der Berechnung von Fakultäten wird sowieso nur ein kleiner Teil der Funktionen benötigt: Multiplikation, Addition und Zuweisung. Diese drei Funktionen sind in der Bibliothek fast optimal implementiert.

Mit den Funktionen der Bibliothek läßt sich die Fakultätsfunktion einfach schreiben als

```
bigint fakultaet;
long i, ende=10000;

bi_const (&fakultaet);

for (i=2; i<=ende; i++)
  bi_mult_long (&fakultaet, i);
```

Eine oft gestellte Frage ist, warum die Zahlen zur Basis 10000 gespeichert werden, und nicht der komplette Zahlenbereich zur Basis 65536 genutzt wird. Der Grund ist einfach: wird die Zahl nicht zu einer Zehnerbasis gespeichert, ist eine Ausgabe im Dezimalsystem relativ unmöglich. Auch die Wahl von 10000 anstatt einer beliebigen anderen Zehnerpotenz ist nicht zufällig. Gängige Prozessoren heutzutage können 32bit Multiplikationen ausführen, und das Produkt 9999*9999=99980001 ist bezüglich dieser Fragestellung das größte, das sich noch in 32 bit darstellen läßt; 99999*99999=9999800001 ist zu groß. Tatsächlich operierte die erste Implementation der Bibliothek auf 8bit Werten zwischen 0 und 99, mit denen 16bit Multiplikationen durchgeführt wurden. Doch in dieser Darstellung werden z.B. bei der Multiplikation zweier Zahlen viermal so viele

Multiplikationen benötigt. Da 32bit Prozessoren eine 32bit Multiplikation fast genauso schnell ausführen wie eine 16bit Multiplikation, wird die Geschwindigkeit beim Übergang von 8bit auf 16bit Werte fast vervierfacht. Auf einer 64bit Architektur mit echter 64bit Multiplikationseinheit wäre eine erneute Steigerung um den Faktor 4 zu erreichen -- einfach durch Anpassung des Datentyps.

Aufwandsabschätzung

Werden zwei Zahlen der Längen l_1 und l_2 multipliziert (Länge = Anzahl der Elemente im Array), so sind hierzu $l_1 \cdot l_2$ Multiplikationen notwendig, und das Ergebnis ist von der Länge $l_1 + l_2$. Die Länge einer Zahl ist hier $l_n = 1 + \log_{10000} n$. Die Länge der Fakultät einer Zahl *n* ist daher die Summe der Länge der Einzelzahlen, $l_{n!} = n + \sum_{i=1}^{n} \log_{10000} i = n + \log_{10000}(n!)$.

Die Anzahl von Multiplikationen zur Berechnung von $n! = n \cdot (n-1)!$ ist also $M_n = (1 + \log_{10000} n)(n - 1 + \log_{10000}((n-1)!)) = (n-1) + (n-2)\log_{10000} n + 2\log_{10000}(n!)$. Die exakte Anzahl von Multiplikationen ist $M_{ges} = \sum_{i=1}^{n} M_i$, deren Formel sich zwar hinschreiben ließe, doch recht nichtssagend wäre.

Bessere Aussagen ergeben sich, wenn zur Abschätzung als Länge jeder primitiven Zahl die hier auftretende Obergrenze 2 eingesetzt wird. Damit wird $l_{n!} = n + \sum_{i=1}^{n} 2 = 3n$, $M_n = 2(3n) = 6n$ und $M_{ges} = \sum_{i=1}^{n} 6i = 3n^2 + 3$. Der Aufwand der Fakultätsberechnung steigt nach dieser Rechnung quadratisch (in der exakten Rechnung steigt der Aufwand ein klein wenig schneller als quadratisch, da dort die Länge der Einzelzahlen nicht konstant ist, sondern langsam, mit dem Logarithmus zur Basis 10000, wächst).

Beim Herumspielen mit den Logarithmen erkennt man auch eine einfache Möglichkeit, die Fakultäten mit bereits vorhandenen Mitteln korrekt, wenn auch ein wenig ungenau, auszurechnen. Ist Z eine Zahl in Exponentialschreibweise, d.h. $Z = (f, e) = f \cdot 10^e$ mit *e* ganzzahlig und *f* einer Fließkommazahl kleiner 1, so ist $\log_{10} Z = e + \log_{10} f$. Betrachten wir den Zehnerlogarithmus einer Fakultät, so können wir schreiben: $\log_{10}(n!) = \sum_{i=1}^{n} \log_{10} i = \sum_{i=1}^{n} (e_i + \log_{10} f_i)$. Wenn wir den Zehnerlogarithmus jeder Zahl nehmen und Vor- und Nachkommaanteil getrennt aufaddieren, erhalten wir den Logarithmus der Fakultät, die nun einfach als $Z = \exp f \cdot 10^e$ gedruckt werden kann. Das Ergebnis ist der korrekte Wert der Fakultät, doch nur auf maximal 20 Stellen genau.

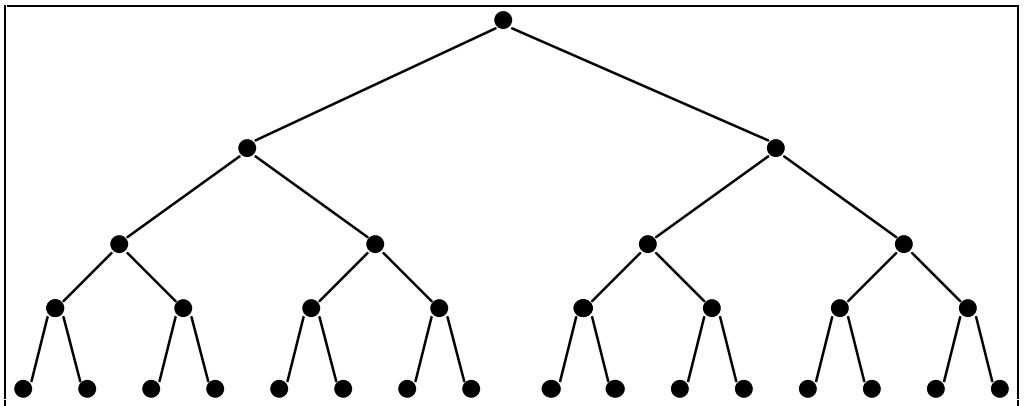
In ein kleines Programm gepackt, ist diese Routine doch sehr hilfreich, um die Größe des Arrays im Datentyp bigint, die ja fest gewählt werden muß, festsetzen zu können.

Parallelisierung

Erste Ansätze

Auf den ersten Blick scheint es, als ließe sich die Fakultätsberechnung genauso einfach aufteilen wie die Integralberechnung der ersten Aufgabe, denn $n! = \prod_{i=1}^{n} i = \left(\prod_{i=1}^{\lfloor n/2 \rfloor} i \right) \bullet \left(\prod_{i=\lfloor n/2 \rfloor + 1}^{n} i \right)$. Nach diesem Prinzip läßt sich die Berechnung beliebig aufteilen. Es ist denkbar, jeden Prozeß einen kleinen Teil der Fakultät berechnen zu lassen, und diese Ergebnisse hinterher im Masterprozeß zu multiplizieren. Eine erste Implementation zeigte jedoch schnell, daß die Lösung unbefriedigend ist: Die Multiplikationen im Master nahmen ca. 90% der Berechnungszeit in Anspruch.

Der zweite Ansatz beruht auf der Erkenntnis, daß die Multiplikation der Teilergebnisse wie in einem Baum strukturiert werden kann:



Grafik 1 - Berechnungsbaum der Strategie 2

Die Einzelergebnisse als Blätter des Berechnungsbaumes werden auf jeder Ebene paarweise zum Ergebnis in der Wurzel des Baumes zusammengesetzt. Der Baum ist binär, da die Multiplikation eine binäre Operation ist. Wird den Slaveprozessen beigebracht, Zahlen multiplizieren zu können, so können die Teilergebnisse jeder Ebene parallel berechnet werden. Doch bringt der Ansatz nicht die erhoffte Besserung, da die finale Multiplikation 50% der Rechenzeit verschlingt.

Der Schlüssel zur Parallelisierung liegt darin, auch noch einzelne binäre Multiplikationen (mit zwei Operanden) parallelisieren zu können.

Parallele Multiplikation

Es wurde bereits erwähnt, wie der Datentyp bigint die Zahlen verwaltet, und wie sich aus den einzelnen Elementen des Feldes die Gesamtzahl berechnen läßt: $A = \sum_{\mathrm{i}=0}^{l_a - 1} 10^{4 \cdot i} \cdot \mathrm{digits}[i]$. Durch Auseinanderziehen läßt sich die Zahl A schreiben als $A = \sum_{i=0}^{a_1 - 1} 10^{4 \cdot i} \cdot digits[i] + \sum_{i=a_1}^{l_a - 1} 10^{4 \cdot i} \cdot digits[i]$. Die Multiplikation zweier Zahlen wird zu: $A \cdot B = \left(\sum_{i=0}^{a_1 - 1} 10^{4 \cdot i} \cdot digits[i] \right) \cdot B + \left(\sum_{i=a_1}^{l_a - 1} 10^{4 \cdot i} \cdot digits[i] \right) \cdot B$. Die Multiplikation wird in zwei Teilmultiplikatio-

nen und eine Addition aufgeteilt. Dabei ist jede der Multiplikationen nur noch halb so groß wie die ursprüngliche, und sie bauen nicht aufeinander auf, d.h. können tatsächlich parallel berechnet werden. Der Aufwand der Addition ist im Vergleich zu den Multiplikationen unerheblich. Die Aufteilung von A in zwei Teilstücke ist willkürlich gewählt; das Prinzip funktioniert immer noch, wenn A bzw. B in beliebig viele Teilstücke aufgeteilt werden.

Auf den Baum projiziert ließe sich nun in der zweiten Ebene von unten jede Multiplikation auf zwei Prozesse aufteilen; in der darüberliegenden auf 4, und die Wurzelmultiplikation auf 16.

Algorithmus

Die Implementation soll natürlich unabhängig von der Prozessanzahl sein. Deshalb scheidet das eben zurechtgelegte Bild des Baumes leider aus, denn nicht jede beliebige Zahl ist eine Potenz von 2. Der Algorithmus wird daher so gewählt, daß er von der Anzahl der Prozessoren völlig unabhängig ist:

① Verschicke an alle Prozesse eine kleine Scheibe der Gesamtaufgabe. Dabei ist die Summe der verschickten Scheiben bewußt kleiner als die Aufgabe, d.h. es bleiben noch Scheiben übrig.

② Wenn ein Slave ein Ergebnis abliefert, überprüfe, ob bereits ein anderes Ergebnis vorliegt. Wenn nein, dann verschicke, soweit vorhanden, eine neue Scheibe an den Slave.

③ Wenn ein anderes Ergebnis vorliegt, so müssen beide Werte multipliziert werden. Verteile diese Multiplikation auf alle Slaves, die gerade keine Aufgabe bearbeiten.

④ Mache solange weiter, bis alle Scheiben der Aufgabe verschickt sind und alle Zahlen multipliziert sind.

Wenn dieser Algorithmus endet, so ist das zuletzt erhaltene Ergebnis das Endergebnis.

Ein Vorteil ist auch, daß maximal ein Ergebnis im Master zwischengespeichert werden muß, denn sofort wenn ein zweites Ergebnis eintrifft, werden sie zur Multiplikation weitergeschickt. Der Speicherverbrauch hält sich somit in Grenzen.

Ein weiteres Stichwort ist, daß der Algorithmus "asynchron" arbeitet. An keiner Stelle wird verlangt, daß Ergebnisse zu einem exakt gleichen Zeitpunkt eintreffen. Das Prinzip läßt sich daher auch in heterogenen Umgebungen einsetzen, in denen die Geschwindigkeit der Einzelprozesse nicht konstant ist.

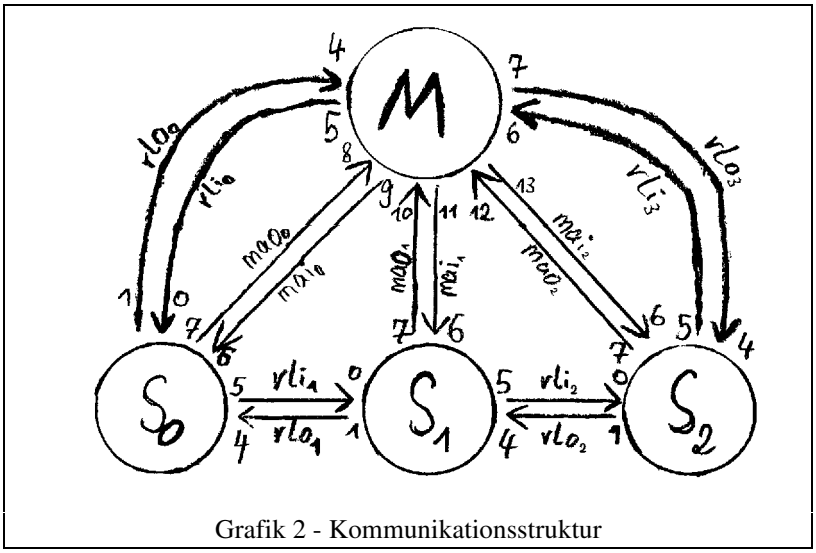
So schön das Verfahren auch ist, es hat zwei Nachteile. Einerseits neigt es zu chaotischem Verhalten; sobald sich die Reihenfolge auch nur zweier Ergebnisse vertauscht, verschieben sich alle folgenden Zeiten, und der Berechnungsbaum kann sich auch bei identischen Anfangswerten völlig unterscheiden. Zweitens arbeitet der Algorithmus nicht immer optimal. Wenn alle Teilintervalle ausmultipliziert sind, muß ein Prozeß stets auf ein zweites Ergebnis zur Multiplikation *warten*. Zusammen mit dem chaotischen Verhalten bedeutet dies, daß auch die Laufzeit kaum vorhersagbar ist, da die Verteilung der Aufgaben zufälligerweise mal mehr und mal weniger optimal ist. Die Laufzeit hält sich innerhalb gewisser Grenzen, kann aber dennoch erheblich schwanken.

Auch die Verteilung der Prozesse nach einem Greedy-Algorithmus (eine Multiplikation wird stets auf alle verfügbaren Prozesse aufgeteilt) knabbert in manchen Fällen an der Optimalität. So kann es passieren, daß erst eine relativ kleine Multiplikation auf wenige Prozesse aufgeteilt wird, und zwei relativ große Ergebnisse müssen kurz darauf mit den zwei einzig freien Prozessen Vorlieb nehmen. Während die kleine Multiplikation schnell erledigt ist, müssen dann alle Prozesse auf die zwei anderen warten. Fatal ist das nur gegen Ende der Berechnung, denn zuvor gibt es immer noch genügend andere Werte zu multiplizieren; erst am Ende, wenn das Ergebnis der zwei Prozesse tatsächlich *benötigt* wird, ist das Warten fatal.

In der Praxis zeigt sich, daß die Wartezeiten auch in schlechten Fällen zwar unangenehm aber erträglich sind. Erwähnt werden sollte natürlich, daß in der zur Verfügung stehenden Zeit kein besserer (oder gar optimaler) Algorithmus erdacht werden konnte, der in gleicher Weise flexibel und frei skalierbar ist.

Kommunikationsstruktur

Es wird haargenau die gleiche Kommunikationsstruktur gewählt wie beim *Game of Life*:



Grafik 2 - Kommunikationsstruktur

Die Struktur hat zwei Vorteile, einerseits ist sie bereits getestet, geprüft und für gut befunden, und andererseits stellt sie eine Obermenge der tatsächlich benötigten Datenkanäle dar. Die direkten Kanäle vom Master zu den Slaves werden verwendet, um die Aufgaben und Multiplikanden zu verschicken, und über den Ring laufen verschiedene Statusmeldungen. Dabei werden nur die unidirektionalen Kanäle gegen den Uhrzeigersinn verwendet, die Rückrichtung liegt brach.

Für die Realisierung der Struktur in einer CDL-Datei sei auf das Protokoll zum *Game of Life*, Kapitel "*CDL-Manie*", verwiesen.

Implementationsfragen

Multiplikation im Eimer

Sobald der Master zwei Werte zur Multiplikation hat, wird die Berechnung auf alle momentan verfügbaren Prozesse aufgeteilt. Dabei wird der größere Multiplikand in entsprechend viele Teilstücke zerlegt, der andere Multiplikand wird in einem Stück gelassen. Auf diese Weise ist die Aufteilung leicht zu implementieren und ist völlig unabhängig von der Prozessanzahl.

Die Einzelergebnisse, die zurückgeliefert werden, werden im Master addiert. Die Addition zweier Zahlen ist unaufwendig und bremst den Master nicht merklich. Es müssen jedoch Speicherbereiche freigehalten werden, in denen die Summation stattfinden kann. Zu diesem Zweck existieren im Master sogenannte Buckets:

```
typedef struct {
    int inuse;
    int numprocs;
    int aktprocs;
    bigint value;
} mulbuck;
```

Die Struktur speichert den jeweiligen Wert des Eimers (zu dem immer dann addiert wird, wenn ein neues Teilergebnis eintrifft), die Anzahl der erwarteten Teile und die Anzahl der bereits eingetroffenen Teile. Wenn `numprocs==aktprocs` ist, kann der Wert aus dem Eimer in die Ergebnisspeicher des Masters kopiert und weiterverwendet werden. Der Master muß eine geeignete Anzahl dieser Eimer bereithalten, denn jede parallele Multiplikation benötigt einen. Wenn keine freien Eimer übrig sind, kann der Master die Multiplikation nicht wie gewünscht verteilen, sondern muß sie an einen Einzelprozessor schicken (denn für eine nicht verteilte Multiplikation wird auch kein Eimer zum Summieren von Teilergebnissen benötigt)

Eine Obergrenze für die Anzahl an benötigten Eimern ist die Hälfte der Prozessanzahl, doch in der Praxis werden deutlich weniger benötigt. Bei 16 Prozessen sind 4 Eimer völlig ausreichend.

Befehle

Alle Befehle bzw. Steuerkommandos werden in der folgenden Struktur versendet:

```
typedef struct {
    int prozess;
    int command;

    struct {
        int anfang;
        int ende;
    } aufgabe;

    struct {
        long length1;
        long length2;
        long first1;
        long first2;
    } multiply;

    struct {
        long length;
        long first;
    } ergebnis;
} comm;
```

Steuerbefehle

Es gibt die folgenden Steuerbefehle, die über den Ring verschickt werden:

COUNT Wird initial zum Zählen der Prozesse im System verwendet.

QUIT Befiehlt den Slaves, sich zu beenden.

POLL Siehe Erklärung zum Kommunikationsdilemma

Aufgaben

Aufgaben werden nicht über den Ring, sondern über die direkten Kanäle an jeden einzelnen Prozeß einzeln verschickt. Die Art der Aufgabe wird durch das Feld `command` bestimmt.

AUFGABE Der Prozeß erhält die Aufgabe, ein Teilintervall der Fakultät zu berechnen: $\prod_{i=anfang}^{enae} i$

MULTIPLY Der Prozeß erhält zwei Zahlen zur Multiplikation. Dabei ist es dem Prozeß vollkommen egal, ob es sich um einen kleinen Teil einer parallelen Multiplikation oder um eine Einzelaufgabe handelt. Die Felder der Unterstruktur `multiply` enthalten die Werte `length` und `first` beider Zahlen (siehe Beschreibung des Datentyps). Im Anschluß an ein Paket diesen Typs werden die beiden Zahlen verschickt, wobei nur der Bereich zwischen `first` und `length` übertragen wird.

RESULT Mit diesem Kommando meldet ein Slave dem Master, daß die Berechnung abgeschlossen und der Slave bereit ist, das Ergebnis zu übertragen. `length` und `first` enthalten die entsprechende Werte des Ergebnis. Diese Art Kommando wird über den Ring verschickt; das Ergebnis selber wird dennoch direkt übertragen.

Übertragung der Zahlen

Zahlen werden stets über die direkten Kanäle zwischen Master und Slave ausgetauscht. Prinzipiell wäre es möglich, einfach die komplette Struktur vom Typ `bigint` zu übertragen. Doch bei kleinen Zahlen wäre dies eine enorme Verschwendung, da nur ein paar Ziffern der Zahl belegt sind, und die komplette Struktur eine beliebige Größe bis hin zu hunderten von Kilobytes haben kann. Doch in der Struktur sind die Werte `first` und `length` zu finden, die auf das erste bzw. letzte belegte Feld in der Zahl zeigen. In dem Paket, das die Zahl ankündigt (entweder vom Typ `MULTIPLY` oder `RESULT`) werden diese beiden Werte übertragen, und anschließend wird nur dieser Teil des Feldes gesendet. Um beim Empfänger die anfangs genannten Bedingungen zu erfüllen, daß alle nicht verwendeten Bereiche der Zahl mit 0 initialisiert sind, muß dieser vor dem Empfang den zuvor belegten Bereich löschen.

Das Kommunikationsdilemma

Als Vorteil des Systems wurde gepriesen, daß es von der Geschwindigkeit der einzelnen Berechnungen unabhängig ist. Das bedeutet für den Master, daß er *nicht weiß*, welcher Prozeß als nächster sein Ergebnis

abliefern wird. Ein solches Problem wird traditionell mit dem Systemaufruf `select(2)` gelöst (dieser Befehl horcht auf vielen Kanälen gleichzeitig, und meldet, wenn auf einem beliebigen Kanal Daten bereit stehen), der unter Helios zwar existiert, aber nicht funktioniert. Ein Polling des Masters auf den Kanälen scheidet ebenfalls aus, da Helios kein nicht-blockierendes Polling unterstützt. Wie aber soll der Master nun wissen, von welchem Kanal er zu lesen hat?

Als Lösung schickt jeder Slave, sobald er seine Berechnung fertiggestellt hat, ein Paket mit `command=RESULT` gegen den Uhrzeigersinn durch den Ring zum Master. Dieser liest das Feld `prozess` im Paket und kann so das Ergebnis ohne zu blockieren direkt aus dem richtigen Kanal fischen.

Auf dem Weg vom Slave zum Master muß das Paket eventuell über andere Slaves hinweglaufen, über Slaves also, die möglicherweise gerade selber mit ihrer Berechnung beschäftigt sind. Damit dies nicht zu einem Hindernis wird, erhält jeder Slave einen zweiten Prozeß, den Kommunikationsprozeß, der parallel zum Rechenprozeß operiert und die Pakete links aus dem Ring entnimmt und rechts wieder hineinstopft. Den Kommunikationsprozeß machen wir auch dafür verantwortlich, die Bereitschaftsmeldung des Slaves zu verschicken, wenn die Berechnung im Rechenprozeß terminiert. Der Slave enthält eine Statusvariable. Wenn eine Berechnung beendet ist, erhält diese im Rechenprozeß den Wert `DONE`. Wenn der Kommunikationsprozeß die Variable mit diesem Wert findet, so wird das entsprechende Paket verschickt, und der Status auf den Wert `IDLE` zurückgesetzt (da der Prozeß nun, bis zum Eintreffen der neuen Aufgabe) nichts mehr zu tun hat.

Doch - wenn der Kommunikationsprozeß auf das Eintreffen eines neuen Paketes links im Ring wartet, so blockiert er in dieser Stellung, und würde den Statuswechsel des Rechenprozesses erst bemerken, wenn zufällig mal das nächste Paket vorbeimarschiert.

Um eine solche Blockade zu vermeiden, wird das eine und genau eine Pollpaket mit `command==POLL` eingeführt. Das Paket hat als einzige Aufgabe, immer wieder und wieder durch den Ring zu laufen. Auf den ersten Blick macht das natürlich überhaupt keinen Sinn, doch dank dieses Paketes werden alle Kommunikationsprozesse von Zeit zu Zeit *aufgeweckt* und erhalten die Möglichkeit, den Statuswechsel des Rechenprozesses zu überprüfen. Auch durchläuft das Paket den Master, der ebenfalls die Möglichkeit erhält, wiederkehrende Aktionen durchzuführen. Doch zu dieser wiederkommenden Aktion kommen wir später.

Implementation

Der Slave

Der Slave ist ein fürchterlich einfaches Programm und besteht aus ein paar Kommunikationsbefehlen und Aufrufen der `bigint` Funktionsbibliothek. Nach dem gewohnten Zähldurchgang, in dem ein Paket zum Zählen der Slaves durch den Ring geschickt wird und der Initialisierung der Zahlen startet der Slave mittels eines `Fork` den Rechenprozeß, während das Hauptprogramm selber in den Kommunikationsprozeß verzweigt. Nun rechnet der Rechenprozeß, und der Kommunikationsprozeß kommuniziert, bis sie den Befehl `QUIT` erhalten und sich

daraufhin verabschieden. Einzig erwähnenswert ist noch die Variable `status`, die, da auf sie von beiden Prozessen schreibend zugegriffen wird, durch eine Semaphore geschützt ist.

Der Master

Der Master ist ungleich komplizierter, da er eine Vielzahl von Bedingungen abfragen und bewältigen muß. Für jeden einzelnen Prozeß wird eine Struktur vom Typ `procstat` allokiert und verwaltet, die ständig den Status jedes Prozesses speichert. Neben rein informativen Feldern merkt sich der Master hier beispielsweise auch, ob die Multiplikation, die der Prozeß durchführt, Teil einer parallelen Multiplikation ist (d.h. daß das Ergebnis in einen Eimer geworfen werden muß). Auch enthält der Master zwei Zahlenspeicher, die Ergebnisse aufnehmen. Es werden nur zwei gebraucht, denn sobald zwei Ergebnisse eingetroffen sind, werden sie sofort wieder zur Multiplikation verschickt.

Die Hauptarbeit wird von einer Handvoll Funktionen geleistet:

`sendaufgabe ()`
: Versendet ein Scheibchen der Gesamtaufgabe, d.h. eine Aufgabe des Typs $\prod_{i=anfang}^{enae} i$.

`sendmultcom ()`
: Versendet eine Multiplikationsaufgabe an genau einen Prozeß, d.h. ohne die Multiplikation zu verteilen.

`distmult ()`
: Versendet eine parallele Multiplikationsaufgabe an alle Prozesse, die gerade frei sind (auch diese Information wird in der `procstat`-Struktur gespeichert). Die erste der beiden Zahlen, die der Funktion als Parameter übergeben werden, wird in ebenso viele gleiche Teilstücke aufgeteilt. Wie Meßwerte später belegen, läuft eine verteilte Multiplikation auf dem homogenen Transputernetz ziemlich optimal ab; alle Einzelprozesse rechnen ungefähr gleich lange an der Aufgabe.

`seldistmult ()`
: Versendet eine Multiplikationsaufgabe. Wenn nur ein Prozeß frei ist, oder wenn selbst die längere der beiden Zahlen zu kurz ist (kleiner 400 Ziffern), so wird die Multiplikation an einen Prozeß verschickt, ansonsten an alle.

`beschaeftige ()`
: Wird aufgerufen, wenn mindestens ein Prozeß gerade frei ist. Je nach Situation wird entweder die Funktion **`sendaufgabe()`** oder **`seldistmult()`** aufgerufen.

`calcloop ()`
: Leistet die wahre Hauptarbeit. In einer großen Schleife, die terminiert, wenn die Berechnung beendet ist und das letzte Ergebnis erhalten wurde, werden die Ergebnisse von den Slaves empfangen, zusammengesetzt und weitergeschickt

Programmbeschreibung

Das Programm benötigt zwei Parameter, zum einen die Zielzahl (also das *n*, dessen *n!* berechnet werden soll), sowie die Scheibengröße. Letztere bestimmt, wie groß ein Teilintervall ist, das zur Berechnung versendet wird. Beide Werte können über die Umgebungsvariablen `ENDE` und `STEP` übergeben werden; falls mindestens eine davon nicht existiert, werden beide Werte von der Tastatur abgefragt.

Sind die Parameter bekannt, beginnt der Master mit der Berechnung. Ein Protokoll der Berechnung wird fortlaufend auf dem Bildschirm gedruckt. Hier ein Auszug aus der Berechnung der Fakultät von 8000 auf 8 Prozessoren.

```
5.59 Calculating     7601 to   8000  on  03
5.80 Got result of   6801 to   7200 from 06   (    1.08 secs)
5.86 Sending multiply     874x384    to  06
6.35 Got result of   7201 to   7600 from 00   (    1.35 secs)
6.86 Got result of   7601 to   8000 from 03   (    1.27 secs)
6.90 Sending multiply     389x386    to Bucket 00 [00,03]
7.07 Got result of mult         705 from 05   (    2.18 secs)
7.13 Got result of mult         931 from 04   (    3.59 secs)
7.18 Sending multiply    1006x755    to Bucket 01 [04,05]
```

Die Zahl am Anfang der Zeile ist die Zeitachse, die die momentane Zeit auf eine hundertstel Sekunde genau (behauptet zumindest Helios) angibt. Der Ausschnitt zeigt also, was sich in gerade einmal 2.5 Sekunden abgespielt hat. Bei `T=5.59` verschickt der Master das Intervall von 7601 bis 8000 zur Multiplikation an Prozeß Nr. 03 (Prozesse werden von 0 an numeriert). Gerade einmal 0.2 Sekunden später trifft ein Ergebnis von Prozeß 06 ein, der den Bereich von 6801 bis 7200 ausmultipliziert hat und dafür ganze 1.08 Sekunden benötigt hat. Da der Master bereits kurz zuvor ein anderes Ergebnis erhalten hatte, schickt er postwendend eine Multiplikation an Prozeß 06 zurück. Die Angabe 874x384 nennt den `length` Parameter beider Multiplikanden, d.h. hier werden zwei Zahlen der Länge 3496 bzw. 1536 Ziffern multipliziert.

Weitere 500 Millisekunden später erhält der Master ein Ergebnis von Prozeß 00. Da zur Zeit keine Multiplikation möglich ist und auch keine Scheiben mehr übrig sind (die letzte wurde ja gerade eben verschickt), muß Prozeß 00 kurz warten, bis auch Prozeß 03 sein Ergebnis abliefert. Jetzt sind zwei Prozesse frei, deshalb wird die Multiplikation auf beide aufgeteilt; beide Einzelergebnisse werden hinterher in Eimer 00 geworfen und dort aufsummiert.

Das komplette Protokoll der Berechnung findet sich im Anhang.

Am Ende der Berechnung wird noch eine Tabelle mit gesammelten Informationen und Werten ausgegeben:

Proc	status	total	calctime	waittime	efficiency	mults
0	24	1200	45.03	3.89	91.86	5
1	24	400	48.43	0.40	98.80	5
2	24	800	47.74	0.90	97.39	5
3	24	1200	45.17	3.38	92.15	5
4	24	400	44.70	3.74	91.19	5
5	24	1200	44.17	4.08	90.11	4
6	24	1200	41.87	6.21	85.41	4
7	24	1600	45.24	2.92	92.29	4

```
Calculation time:      49.02 secs
Spent waiting:          6.51%
Spent calculating:     92.40%
```

Die Spalte `total` listet die Summe der Größe der Scheiben, die der Prozeß berechnet hat, und `mults` listet die Summe durchgeführter Multiplikationen. Für jeden Prozeß wird die Wartezeit und Rechenzeit sowie die daraus berechnete Effizienz ausgebegeben. Die Zeiten werden weiterhin aufsummiert und unten noch einmal global gedruckt. Hier wurde eine globale Effizienz von 92% erreicht, was ein recht guter Wert ist. Prozeß 01 und 02 sind sogar kaum zum Verschnaufen gekommen.

Der Ausdruck des Ergebnisses dieser Berechnung, der Fakultät von 8000, befindet sich im Anhang.

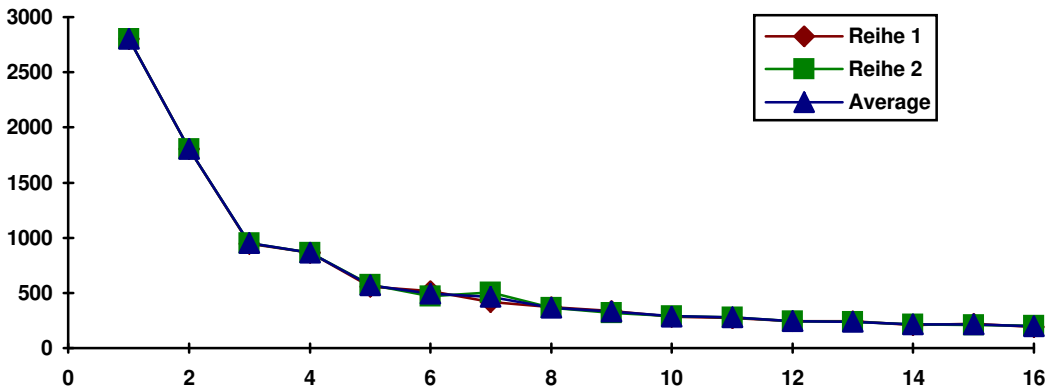
Performance

Für die Geschwindigkeitsmessungen wird jeweils die Fakultät von 20000 (knapp 80000 Ziffern) berechnet. Zwei Meßreihen wurden durchgeführt. Bis 8 Prozessoren fanden die Messungen auf der Architektur *cube* statt, die darüberliegenden auf *4x4*. Alle Zeitangaben sind in Sekunden.

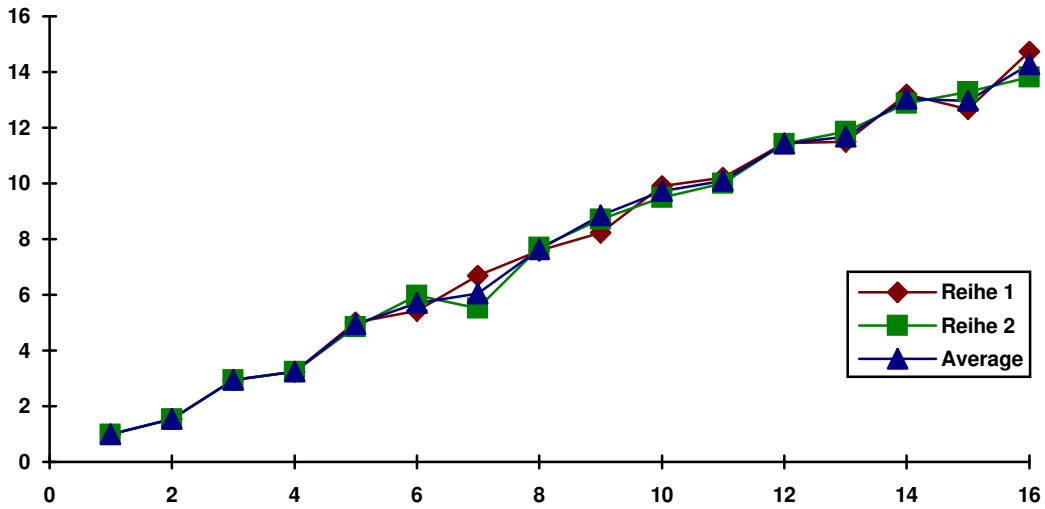
Tabelle

	Meßreihe 1			Meßreihe 2			Durchschnitt		
Prozessoren	Zeit	SpdUp	Effiz	Zeit	SpdUp	Effiz	Zeit	SpdUp	Effiz
1	2806.0	1.000	1.000	2806.0	1.000	1.000	2806.0	1.000	1.000
2	1806.3	1.553	0.777	1805.8	1.554	0.777	1806.0	1.554	0.777
3	954.4	2.940	0.980	954.4	2.940	0.980	954.4	2.940	0.980
4	867.3	3.235	0.809	867.1	3.236	0.809	867.2	3.236	0.809
5	558.3	5.026	1.005	576.6	4.866	0.973	567.5	4.945	0.989
6	516.1	5.437	0.906	469.0	5.983	0.997	492.6	5.697	0.949
7	419.3	6.692	0.956	508.7	5.516	0.788	464.0	6.047	0.864
8	370.2	7.580	0.947	364.7	7.694	0.962	367.5	7.636	0.955
9	340.5	8.241	0.916	322.2	8.709	0.968	331.4	8.468	0.941
10	283.3	9.905	0.990	293.4	9.564	0.956	288.4	9.731	0.973
11	275.0	10.20	0.928	280.8	9.993	0.908	277.9	10.10	0.918
12	245.0	11.45	0.954	245.8	11.42	0.951	245.4	11.43	0.953
13	244.0	11.50	0.885	236.7	11.85	0.912	240.4	11.67	0.898
14	212.7	13.19	0.942	218.0	12.87	0.919	215.4	13.03	0.931
15	221.5	12.67	0.844	211.2	13.29	0.886	216.4	12.97	0.865
16	190.4	14.74	0.921	203.1	13.82	0.863	196.8	14.26	0.891

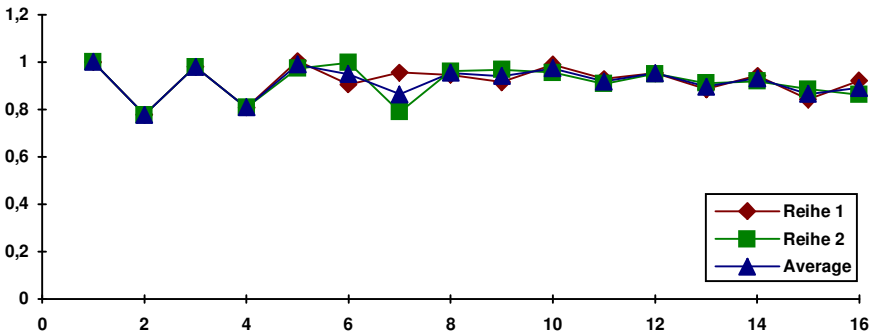
Grafiken



Grafik 3 - Laufzeit des Programms



Grafik 4 - Speed - Up



Grafik 5 - Effizienz

Diskussion

Betrachtet man die Spitzenwerte des Programms, muß man sehr zufrieden sein, da es doch in vielen Fällen knapp die optimale Performance erreicht (bzw. sogar in einem Fall darüber liegt). Leider wird das Ergebnis durch einige Ausrutscher gestört. An mehreren Meßpunkten liegt die Effizienz teilweise deutlich unter 90%. Dieser Effekt liegt in dem Algorithmus begründet, der letztendlich ein Greedy-Algorithmus ist und stets nur die lokal beste Lösung sucht. Betrachten wir einen Ausschnitt aus der Logdatei für 2 Prozessoren:

```
   507.51 Sending multiply    9441x6693   to  00
...
   575.71 Sending multiply    2663x536    to  01
   598.18 Got result of mult        3013 from 01   (   22.47 secs)
  1427.89 Got result of mult       15072 from 00   (  920.38 secs)
  1427.89 Sending multiply   16134x3200   to Bucket 00 [00,01]
```

Zum Zeitpunkt 507.51 erhält der Prozeß 00 eine riesige Multiplikation. Knappe 100 Sekunden später hat Prozeß 01 alle restlichen Aufgaben erledigt. Übrig bleibt nur noch die letzte Multiplikation zwischen dem Ergebnis beider Prozesse zum Endergebnis. Doch da diese erst beginnen kann, wenn auch 00 das Zwischenergebnis liefert, muß 01 so lange warten: glatte 830 Sekunden, was natürlich ins Gewicht fällt.

Auffallen tun auch die beiden Werte für 7 Prozessoren, die das chaotische Verhalten des Systems zeigen. Trotz völlig identischer Anfangsbedingungen unterscheidet sich die Berechnungszeit erheblich. Insgesamt ist an den Graphen kein eindeutiger Trend erkennbar. Nur mit Mühe ein zaghaftes Absinken der Effizienz bei vielen Prozessoren erkennen.

Die Frage, die man sich natürlich stellen muß ist, ob die "Fehler" auch *vermeidbar* sind. Selbst in dem obigen einen Einzelfall fällt es schwer, die Berechnungsreihenfolge von Hand so zu vertauschen und verdrehen, daß ein besserer Wert entsteht. Und solange nicht ein solches Prinzip gefunden wird, das sich weiterhin auf beliebig viele Prozesse skalieren läßt, sollte man sich mit dem implementierten Algorithmus zufrieden geben.

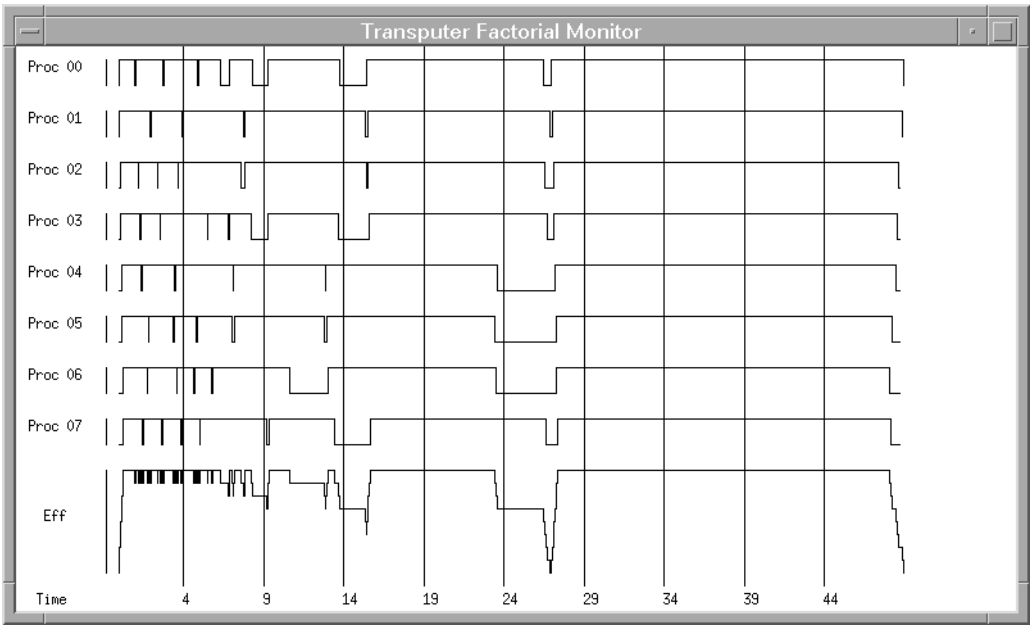
Durch vermehrte Messungen müßte, insbesondere bei höherer Prozessanzahl, anhand deutlich mehr Messungen festgestellt werden, welche Fälle die Regel und welche Fälle die Ausnahme sind. Aus einer großen Anzahl Messungen muß hier ein Mittelwert gebildet werden, anders läßt sich ein solcher Prozeß nicht beschreiben.

Vielleicht auch noch ein Wert zur Korrektheit des Programmes. Das Programm druckt Zahlen aus, die noch nie ein Mensch zuvor gesehen hat. Wer garantiert, daß das Ergebnis auch stimmt? Zum Glück kann man die Werte mit sich selbst vergleichen, beziehungsweise mit der sequentiellen Implementation. Denn die parallele Version setzt sich die Zahlen durch völlig andere Funktionaufrufe zusammen. Das Ergebnis wird von fast völlig anderem Programmcode erzeugt, und dennoch stimmen die Werte in jeder Ziffer überein.

Weitere Sicherheit verspricht der Vergleich der Ausgabe mit der Abschätzung über die Logarithmengesetze, die anfangs genannt wurde, die zumindest die Anzahl an Ziffern und das erste Dutzend Ziffern der Zahl liefert. An der Genauigkeit des Ergebnisses kann somit kaum noch Zweifel angemeldet werden.

X-Window-Ausgabe

Das schönste zum Schluß. Für denjenigen, dem die Zahlenkolonnen in der Logdatei zu langweilig sind, wurden, auf der RX11-Bibliothek basierend ein paar Monitorfunktionen in den Master eingebaut. Beim Berechnungsstart versucht der Master, sich zum RX11-Server zu verbinden. Klappt das nicht, bleibt alles bei der Textausgabe. Wird eine Verbindung hergestellt, so öffnet sich ein Fenster, in dem man den Status der einzelnen Prozesse und des Gesamtsystems komfortabel einsehen kann.



Hier zeigt der Monitor den Status des Programms während der Berechnung der Fakultät von 8000 auf 8 Prozessoren. Die oberen 8 Zeilen zeigen den Status der einzelnen Prozesse (beschäftigt bzw. idle), die letzte Zeile zeigt die Gesamteffizienz des Systems (beschäftigte Prozesse durch Anzahl der Prozesse). Man erkennt sehr schön die hektische Anfangszeit. Hier ist kein Prozeß lange idle, da noch genügend Scheiben vorhanden sind, die berechnet werden wollen. Erst als nach bereits 9 Sekunden alle Scheiben berechnet sind, passiert es, daß manche Prozesse warten müssen. Kurz vor 24 Sekunden sieht man die Prozesse 04, 05 und 06 abliefern und ca. 3 Sekunden warten, bis sie in die große Endmultiplikation einbezogen werden, die glatte 20 Sekunden in Anspruch nimmt, bis die Berechnung gegen 49 Sekunden beendet ist.

Zum Vergleich sei nochmal auf die ausgedruckte Logdatei eben dieser Berechnung im Anhang verwiesen.

Epilog

Ein Wunsch bei der Berechnung ist natürlich, daß auch möglichst große Werte berechnet werden können. Doch auf den Transputern reichte leider der Speicher nicht, um der bisherigen Bestleistung nahe zu kommen.

Als Experiment wurde die parallele Implementation auch auf die Verwendung von TCP Sockets portiert, um so auf beliebigen Unix-Workstations laufen zu können. Hier wird zunächst der Master gestartet, zu dem sich dann eine beliebige Anzahl von Slaves aktiv verbinden kann; auch während des Betriebes kann die paral-

lele Rechnungsmaschine weiter wachsen. Verteilt auf über 70 Rechner des RBI, der TI und der Telematik ließ sich nun die Fakultät von 200000, die zuvor in 3 Tagen berechnet wurde, fast im Handumdrehen drucken. Nach diesem erfolgreichen Testlauf wurde die nächste Grenze durchbrochen und die Fakultät von 1000000 berechnet. Um die Zahlen überhaupt speichern zu können, benötigte jeder Slave 12, der Master gar über 40 Megabyte Speicher. Auch die benötigte Zahl von 70 offenen Filedeskriptoren für den Master mußte erfüllt werden. Nach exakt 165579 Sekunden (46 Stunden) stand das Ergebnis fest, und die 5,56 Millionen Ziffern konnten gedruckt werden.

Zu schaffen machte allerdings die Heterogenität des Netzwerkes, da sich die (relative) Rechenleistung der einzelnen Rechnern um mehr als den Faktor 4 unterschied. Beim Starten eines jeden Slaves wurde automatisch die Fakultät von 2000 als "Geschwindigkeitsindex" berechnet. Bei verteilten Multiplikationen wurde die so ermittelte Geschwindigkeit verwendet, um dem Prozeß ein mehr oder weniger großes Stück der Aufgabe zuzuteilen. Dennoch funktionierte die Verteilung offensichtlich nicht fehlerfrei, da die Berechnung im Endeffekt von zwei Rechnern gebremst und nur eine Effizienz von ca. 50% erreicht wurde.

Es bleibt also noch viel Arbeit bis zu einer optimalen Implementation. Einerseits muß das heterogene Verteilungsproblem besser gelöst werden (ein Gedanke besteht darin, daß laufende Multiplikationen unterbrochen und auf mehr Prozesse verteilt werden können); und zweitens wäre es ideal, die parallele Maschine so flexibel zu gestalten, daß sich im laufenden Betrieb neue Rechner anmelden und auch abmelden können. So wäre es denkbar, daß jeder Rechner überwacht wird: ist er frei, so wird dort ein neuer Slave gestartet, ist er besetzt, beendet sich ein laufender Slave, um die Benutzer nicht zu stören. Damit der Master nicht selber mit allen Slaves direkt kommunizieren muß, würden Cluster eingeführt, d.h. der Master vergibt Aufgaben an einen Mittler, der die Aufgabe auf die ihm unterstehenden Slaves weiterverteilt.

Eine solche Weiterentwicklung ist jedoch nicht geplant, einerseits wegen dem programmtechhnischen Aufwand, der für eine solch absolut sinnlose Aufgabe einfach viel zu schade ist, und andererseits steigen die Anforderungen an den Rechner, der den nicht migrationsfähigen Master beherbergt, in astronomische Höhen. Zur Berechnung von 2000000! ist im Master ein Speicherverbrauch von 100 Megabytes zu erwarten.

Und für den denjenigen, dem selbst solche Zahlen noch zu klein sind, gibt es ja das Programm zum Drucken eines *Googolplex* ...