Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang

Abgabe: 28. November 2013 in der Übung

Übung 10 Bakery-Algorithmus (5 Punkte)

In der Vorlesung wurden verschiedene Möglichkeiten vorgestellt, um den wechselseitigen Ausschluß zu realisieren (Spin Locks, Ticketing, ...). Folgender Algorithmus in unserer abstrakten Notation realisiert den wechselseitigen Ausschluss ohne spezielle Maschineninstruktionen für P Prozesse.

```
1  parallel bakery
2  {
3    const int P            = 8;
4    int       number[P]    = {0[P]};
5    int       choosing[P]  = {0[P]};
6
7    process Proc [int i in {0,...,P-1}]
8    {
9      int mine;
10     while (true)
11     {
12       // entry protocol
13       choosing[i] = 1;
14       mine        = 0;
15
16       for (int k=0; k<P; k++) mine = max(mine,number[k]);
17       number[i]   = mine+1;
18       choosing[i] = 0;
19
20       for (int k=0; k<P; k++)
21       {
22         while (choosing[k]);
23         while (number[k]!=0 &&
24             (number[k]<number[i] || (number[k]==number[i] && k
     <i)));
25       }
26
27       // critical section follows here
28       // exit protocol
29       number[i] = 0;
30       // uncritical section follows here
31     }
32   }
33 }
```

Erklären Sie die Funktionsweise des Algorithmus (Tipp: Denken Sie an den *Ticket-Algorithmus*). Überlegen Sie, warum zwei Prozesse nicht gleichzeitig in den kritischen Abschnitt gelangen können, sofern man sequentielle Konsistenz des Speichers annimmt. Diskutieren Sie, ob der Algorithmus Verklemmungsfreiheit und Schließliches Eintreten garantiert (kein Beweis nötig!).

Übung 11 Goldbach-Theorem mit OpenMP (10 Punkte)

Nach einem unbewiesenen Satz („Goldbach-Theorem“) läßt sich jede gerade Zahl $p \geq 4$ als Summe zweier Primzahlen darstellen (wobei die 1 nicht als Primzahl gezählt wird). Eine einfache Realisierung des Tests wäre beispielsweise:

- Ein Feld $1 \ldots N$ speichert einen `bool`-Wert, ob der Feldindex $i \in 1 \ldots N$ eine Primzahl ist.
- Bei Iteration mit i über das Feld ergibt der Test auf gleichzeitige Primheit für i und $N-i$, ob das Zahlenpaar $(N-i, i)$ die Bedingung erfüllt: Da $i+(N-i) = N$ ist gegebenfalls eine Darstellung durch eine Primzahlsumme gefunden.

Hierzu benötigt man eine Funktion, die bestimmt (und im Feld speichert), ob i eine Primzahl ist:

```
1 inline bool is_prime(long i)
2 {
3   const long j = (long) std::sqrt(i);
4   for (long k=2; k<j+1; ++k)
5     if (i % k == 0)
6       return false;
7
8   return true;
9 }
```

Es gibt hierfür natürlich noch bessere Alternativen, Stichwort *Sieb des Eratosthenes*.

Schreiben Sie ein Programm, das für alle Zahlen i bis zu einer Obergrenze N die Anzahl der möglichen Darstellungs-Summen zählt. Das Programm soll mit OpenMP parallelisiert werden, d. h. jeder Thread soll einen Teil der Zahlen überprüfen und die Anzahl der Darstellungen zählen. Es gibt natürlich mehrere Möglichkeiten der Umsetzung. Es könnte ein globales, von allen Threads zu lesen und zu schreibendes Feld geben, dass die Primheit der Zahlen von 1 bis N speichert; oder jeder Thread speichert sein eigenes Feld. Alternativ kann man die Primheit einer Zahl auch *On the fly* für jedes i mit obiger Funktion ausrechnen.

Messen Sie für Ihr sequentielles (ein Thread) und paralleles (mit 2, 4 und 8 Threads) Programm die Laufzeit in s. Hierzu können Sie die bekannte OpenMP-Zeitmessung verwenden. Wieviel Speed-Up erreichen Sie parallel? Messen Sie für $N = 100$ bis mindestens $N = 10^5$ mit aureichend vielen Messpunkten. Mitteln Sie über mehrere Messungen, um Ausreißer zu vermeiden, und geben Sie auch an, welche Optimierungsstufe Sie verwendet haben. Da der Rechenaufwand exponentiell steigen dürfte, ist die Lastverteilung inhomogen. Testen Sie daher auch die Scheduling-Parameter `static`, `guided` und `dynamic`. Diskutieren Sie Ihre Ergebnisse an Hand von Abbildungen der Rechenzeit über Problemgröße.