Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

Abgabe: 10. November 2009 in der Übung

Übung 6 Fat tree-Netzwerke (5 Punkte)

Bei einem *Fat tree*-Netzwerk erhöht sich die Anzahl der Verbindungen zweier Knoten, je näher man an die Wurzel kommt: Die Blätter des Baumes sind mit einer Leitung mit ihren Vätern verbunden, die nächsthöhere Ebene hat zwei Verbindungen, dann vier usw. Geben Sie für diese Topologie den Knotengrad, die Gesamtzahl der Verbindungen, den Netzwerkdurchmesser und die Bisektionsbreite an.

Übung 7 Bakery-Algorithmus (5 Punkte)

In der Vorlesung haben Sie verschiedene Möglichkeiten kennengelernt, den wechselseitigen Ausschluss zu realisieren (Spin Locks, Ticketing, ...). Folgender Algorithmus realisiert den wechselseitigen Ausschluss ohne spezielle Maschineninstruktionen für P Prozesse. Er wird *Bakery-Algorithmus* genannt und wurde von Leslie Lamport entwickelt:

```
parallel bakery{
  const int P            = 8;
  int       number[P]    = {0[P]};
  int       choosing[P]  = {0[P]};

  process Proc [int i in {0,...,P-1}]{
    int mine;
    while (true){
      // entry protocol
      choosing[i] = 1;
      mine        = 0;

      for (int k=0; k<P; k++) mine = max(mine,number[k]);
      number[i]   = mine+1;
      choosing[i] = 0;

      for (int k=0; k<P; k++){
        while (choosing[k]);
        while (number[k]!=0 &&
            (number[k]<number[i] || (number[k]==number[i] && k<i)));
      }

      // critical section follows here
      // exit protocol
      number[i] = 0;
      // uncritical section follows here
    }
  }
}
```

Erklären Sie die Funktionsweise des Algorithmus (Tipp: Denken Sie an den *Ticket-Algorithmus*). Überlegen und diskutieren Sie, warum zwei Prozesse nicht gleichzeitig in den kritischen Abschnitt gelangen können, sofern man sequentielle Konsistenz des Speichers annimmt.

Übung 8 OpenMP **(10 Punkte)**

In Übung 2 haben wir zwei (quadratische) Matrizen miteindander multipliziert. Wir wollen dieses Programm mit *OpenMP* parallelisieren und die Skalierbarkeit des Problems hinsichtlich der Anzahl der Threads untersuchen. Schreiben Sie dazu ein Programm, das die Matrizen-Multiplikation *OpenMP*-parallel ausführt. Die äußerste `for`-Schleife des Multiplikations-Algorithmus soll dabei von je einem Thread bearbeitet werden, d. h. jeder Zeilendurchlauf wird in einem eigenen Thread gestartet. Allgemein sollen Sie in dieser Übung Rechenzeiten über der Problemgröße oder der Anzahl der beteiligten Threads messen. Hinweise zur Zeitmessung stehen am Ende der Aufgabe.

(a) Verwenden Sie `static`-Scheduling und nehmen Sie jeweils eine Rechenzeit-Meßreihe für $m = 2^n,\ n \in 0, 1, \ldots, 11$ beteiligte Threads für die Matrix-Größen (Grundseite der Matrix) $N = 128, 256, 512$ auf. Plotten Sie ein Zeit-m-Diagramm für jede Problemgröße. Diskutieren Sie, wann sich der Overhead durch die Parallelisierung lohnt, und wann nicht.

(b) Verwenden Sie nun eine Matrix mit $N = 500$ und $m = 1, 2, 4, 8, 16$. Plotten Sie Zeit-m-Messreihen je für `static` und `dynamic`-Scheduling mit jeweils der chunk-size 102 und 13. Gibt es Unterschiede zwischen den Kurven mit verschiedenem Scheduling? Falls ja, wie interpretieren Sie diese?

(c) Konstruieren Sie einen Fall, in dem Sie das *static*-Scheduling verwenden, und in dem die Threads unterschiedlich viel Arbeit zu leisten haben. Eine Idee wäre eine *OpenMP*-parallelisierte Loop über eine streng monoton wachsende Problemgröße, etwa $N = 2^i,\ i = 2, 3, 4, \ldots$, wobei die eigentliche Matrixmultiplikation nicht parallelisiert ist. Manche Threads multiplizieren kleine, andere grosse Matrizen miteinander. Um die den Effekt der ungleichen Lastverteilung zusätzlich zu verstärken ohne Speicherplatz zu vergeuden, führen Sie die Multiplikationen mehrmals aus. Messen Sie die Zeiten und vergleichen Sie dann mit *dynamic*-Scheduling. Bei wirklich inhomogener Lastverteilung würde man einen sichtbaren Vorteil des *dynamic*-Scheduling erwarten. Sie können natürlich auch eigene Ideen realisieren, etwa Matrizen unterschiedlichen Datentyps (`int, double`) in einer *Loop* oder etwas ganz Anderes. Wichtig ist nur die unsymmetrische Last.

Hinweise:

- Auf *nix-Systemen können Sie *OpenMP*-Programme mit dem gcc-Compiler ab Version 4.1.2 unter Verwendung der Option `-fopenmp` kompilieren. Wird diese Option weggelassen, wird das Programm zur seriellen Ausführung übersetzt. Zur Verwendung von *OpenMP* muss die Datei `omp.h` inkludiert werden:

```
#ifdef _OPENMP
  #include <omp.h>
#endif
```

  Die Anzahl der Threads können Sie über die Umgebungsvariable `OMP_NUM_THREADS` in der Shell setzen, oder im Programm durch Aufruf der Funktion `omp_set_num_threads(int m)`. Auf der Vorlesungs-Homepage finden Sie ein kleines Programm, das den grundlegenden Aufbau eines *OpenMP*-Programms illustriert.

- Für die Zeitmessung können wir nicht die Timing-Funktionen aus `timer.h` verwenden, da diese die CPU-Zeit und damit die summierte User-Time *aller* Threads messen würde. Daher messen wir die verstrichene Echtzeit des Haupt-Threads mit der *OpenMP*-Methode, z.B.

```
double tstart = omp_get_wtime();
  // tue was mit mehreren Threads
double tend   = omp_get_wtime();
```

  mit der Einheit s. Hierbei sollten Sie sicherstellen, dass die Zeitmessung nicht zu sehr durch andere Prozesse gestört wird. Um ausreichend viele Floating Point Operation auszuführen, sollten sie bei Bedarf mehrere Iterationen ausführen.