Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, J. Pods

Abgabe: 27. November 2012, 12 Uhr, per E-mail an jurgis.pods@iwr.uni-heidelberg.de

---

Übung 8 Fat-tree-Netzwerke (5 Punkte)

Wir betrachten ein statisches *Fat-tree*-Netzwerk. Diese Netzwerktopologie ist ein Baum, bei dem sich die Anzahl der Verbindungen zweier Knoten erhöht, je näher man an den Wurzelknoten kommt: Die Blätter des Baumes sind mit einer (hier der Einfachheithalber Duplex-) Leitung mit ihren Vätern verbunden, die nächsthöhere Ebene hat zwei solche Verbindungen, dann vier usw. Die Topologie ist in Abbildung 0.3 schematisch dargestellt. Geben Sie für diese Topologie den Knotengrad, die Gesamtzahl der Verbindungen, den Netzwerkdurchmesser und die Bisektionsbreite an.

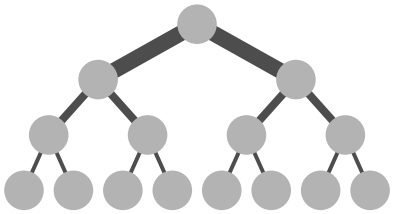

Abbildung 0.3: Statisches Fat-tree-Netzwerk, schematisch. Quelle: Wikipedia

Übung 9 OpenMP: Matrix-Multiplikation (10 Punkte)

In Übung 4 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. Allgemein messen wir in dieser Übung Rechenzeiten und FLOPs über der Problemgröße oder der Anzahl der beteiligten Threads. Hinweise zur Zeitmessung stehen am Ende der Aufgabe.

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

(b) Untersuchen und diksutieren Sie, wie sich Ihr Programm mit `dynamic`-Scheduling verhält.

(c) Modifizieren Sie Ihr Programm, so daß durch verschachtelte parallele Blöcke auch die Instruktionen der inneren Schleife parallel ausgeführt werden, d. h. es wird nun auch für jedes Skalarprodukt innerhalb eines Zeilendurchlaufs ein Thread gestartet. Zur geschachtelten Parallelisierung gibt es die OpenMP-Funktion `omp_set_nested(int k)`, die die Tiefe k der möglichen Schachtelungsebenen setzt. Für jede Ebene muss eine eigene `#pragma omp parallel`-Umgebung gesetzt werden. Fertigen Sie eine Rechenzeit-m-Messung wie oben mit $m \in 1, 2, 4, \ldots, 128$ für $N = 256$ an bei statischem Scheduling. Lohnt sich der zusätzliche Overhead dieser Variante?

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 kann über die Umgebungsvariable `OMP_NUM_THREADS` in der Shell oder im Programm durch Aufruf der Funktion `omp_set_num_threads(int m)` gesetzt werden. 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.