Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang

Abgabe: 16. Januar 2014 in der Übung

Übung 21 MPI: Kommunikationszeiten (5 Punkte)

Wir wollen die Zeiten messen, die bei den MPI-Sende-Routinen `MPI_Send`, `MPI_Ssend`, `MPI_Bsend` und `MPI_Rsend` (synchrones bzw. asynchrones Senden, ...) benötigt werden, bis

1. der aufrufende Knoten wieder die Kontrolle über das Programm nach Beenden des jeweiligen Sende-Befehls erlangt hat,
2. der Empfänger die Nachricht ganz erhalten hat.

Teilaufgabe (a)

Schreiben Sie dazu ein Programm, das per Kommandozeilen-Parameter eine dieser Routinen zum Versenden einer Nachricht an einen anderen Prozess wählt. Die Größe der Nachricht soll ebenso als Programmparameter übergeben werden können. Der Sender mißt die Zeit, die es dauert, bis das Send-Kommando abgeschlossen ist, d.h. er wieder die Kontrolle über das Programm erhält, und der Empfänger die Zeit, die er benötigt, die Nachricht vollständig zu empfangen. Um zuverlässige Werte zu erhalten, sollte es möglich sein, diese Prozedur mehrmals hintereinander auszuführen und die Zeiten dann zu mitteln. Die Anzahl Wiederholungen sollte an die Dauer der Nachrichtenübermittlung angepasst sein, d.h. schnelle Übertragungen werden häufiger wiederholt. Das ganze muss natürlich nicht automatisch geschehen.

Messen Sie für jede Sende-Art die Zeiten mit mehreren Nachrichtengrößen. Die Nachrichtengröße soll dabei von wenigen Byte bis zu einigen MBytes variieren. Messen Sie auch die Hopzeit, d.h. die Übertragungszeit eines einzelnen Bytes.

Teilaufgabe (b)

Die Routine `MPI_Send` kann blockierend oder nicht-blockierend arbeiten. Überlegen Sie sich einen Test, wie herausgefunden werden kann, ab welcher Nachrichtengröße n synchron gesendet wird. Implementieren Sie den Test und finden Sie das passende n im Pool heraus.

Freiwillige Zusatzaufgabe

Wiederholen Sie die Messungen aus Teilaufgabe (a), arbeiten Sie aber nur lokal auf einem Rechner arbeiten (wie das geht, siehe Übung 20). Wie verhalten sich die Zeiten nun?

Hinweise

- Da wir auch den Overhead durch die Kommunikation messen wollen, messen wir die Echtzeit unter Verwendung von `MPI_Wtime()`. Im Pool hat `MPI_Wtime()` die Auflösung $1\mu s$ (Abfrage über `MPI_Wtick()`, was für unsere Zwecke ausreichen sollte.
- Synchronisieren Sie die Prozesse vor dem Starten der Zeitmessung mittels einer `MPI_Barrier()`.
- Profiling und Zeitmessung für MPI-Programme ist nicht trivial. Hier sind einige Links auf hilfreiche alte und neue Dokumente:
  - Ein Beispiel wie wir es in Teilaufgabe (a) implementieren wollen:
    http://cpansearch.perl.org/src/JOSH/Parallel-MPI-0.03/contrib/perf/mpi_timing.c

– Pitfalls in der Zeitmessung:
  http://www.mcs.anl.gov/research/projects/mpi/mpptest/hownot.html
– MPI-Profiling Suite:
  http://www.mcs.anl.gov/research/projects/mpi/mpptest/
– Ein recht ausführliches Paper:
  http://perlplexity.org/Publications/hoefler-collmea.pdf

Übung 22 MPI: Matrixmultiplikation (10 Punkte)

In den Übungen 4 und 9 haben wir zwei (quadratische) Matrizen A, $B \in \mathbb{R}^{n \times n}$ miteinander multipliziert ($C = A \cdot B$) und die Flop-Raten mit Tiling bzw. OpenMP gemessen. Wir wollen das nun unter Verwendung von MPI wiederholen und die Skalierbarkeit des Problems hinsichtlich der Anzahl der Prozesse untersuchen.

Aufgabe

Schreiben Sie ein Programm, das die Matrizen-Multiplikation mit MPI parallel ausführt. Auf der Homepage steht eine sequentielle Variante inklusive Tiling bereit (Übersetzen mit `g++ -fopenmp mm_vanilla.c`), die Sie natürlich nicht verwenden müssen. Sie können sich am Cannon-Algorithmus orientieren (wird in der Vorlesung behandelt, siehe auch das Skript), dürfen aber auch eine sinnvolle andere Strategie implementieren (erklären Sie aber kurz Ihr Konzept). Die Matrix soll aber initial so auf die Prozessoren verteilt sein, dass nicht jeder Prozessor die Ausgangs-Matrizen voll kennt (sonst fällt ja ein Großteil des Kommunikations-Overheads weg). Initialisieren Sie die Matrizen so, daß sie vollbesetzt sind, etwa $a_{ij} = b_{ij} = i+j,\ i,j = 0 \ldots n-1$. Messen Sie die Rechenzeit in Sekunden und die FLOP-Rate für $m = 1, 2, 4, 6, 8, 12, 16, 20, 25, 32$ beteiligte Prozesse und Matrix-Größen (Grundseite der Matrix) $n = 256, 512, 1024, 2048$. Zur Messung mit $m = 1$ können Sie das bereitgestellte sequentielle Programm verwenden. Nehmen Sie zur Zeitmessung wie gehabt `MPI_Wtime()` und synchronisieren vor Beginn der Zeitmessung mittels einer `MPI_Barrier()`. Plotten Sie Diagramme mit der Rechenzeit über den Problemgrößen sowie des Speed-Ups über der Prozessoranzahl P für die verschiedenen Problemgrößen. Diskutieren Sie kurz die Ergebnisse. Falls noch vorhanden, können Sie auch mit den alten Ergebnissen der Aufgaben 4 und 9 vergleichen.

Freiwillige Zusatz-Aufgabe

Verwenden Sie auf den einzelnen Rechenknoten zusätzlich ein Tiling, um den Cache besser auszunutzen, und vergleichen Sie die Messungen mit den Ergebnissen der ungekachelten Variante.

Hinweise

Man könnte daran denken, den Cannon-Algorithmus zur Lösung dieser Aufgabe zu verwenden. Dieser Algorithmus ist aber hier nicht ohne Weiteres möglich: Zunächst einmal muss bei Cannon gelten $N \mod \sqrt{P} = 0$, was einige der oben geforderten Kombinationen unmöglich macht: Die Anzahl der Rechenknoten muss dem Quadrat einer natürlichen Zahl entsprechen. Außerdem ist der Cannon-Algorithmus vor Allem dann effizient, wenn man große Matrizen multipliziert, die nur verteilt gespeichert werden können. In unserer Übung sind die Matrizen aber maximal $2048 \cdot 2048 \cdot 8$ Bytes $= 32$ MBytes groß und passen somit problemlos in den Speicher jedes Knotens. Wenn Sie dennoch den Cannon-Algorithmus umsetzen wollen, passen Sie P und N den Anforderungen an und gehen Sie von Matrix-Größen aus, die nur noch verteilt gespeicher werden können.