Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

Abgabe: 01. Dezember 2009 in der Übung

Übung 14 MPI-Deadlocks (10 Punkte)

Kollektive Kommunikations-Operationen wie etwa Broadcasts können synchronisierend wirken, bei falscher Reihenfolge aber auch zu Deadlocks führen. Manchmal hängt dies davon ab, ob Systempuffer zur Zwischenspeicherung von Nachrichten verwendet werden oder nicht. In der Regel bestimmt dies das Laufzeitsystem und somit die konkrete MPI-Implementierung. Wir betrachten in dieser Aufgabe Fälle, in denen verschiedene Systeme unterschiedliches und teils unerwünschtes Verhalten zeigen würden.

Teilaufgabe (a) (5 Punkte)

Das folgende Listing zeigt zwei `MPI_Bcast()`-Operationen in unterschiedlicher Reihenfolge, die je Implementierung des Laufzeitsystems zu unterschiedlichen Fehlern führt:

```
switch (my_rank) {
  case 0: MPI_Bcast(buffer1, count, type, 0, comm);
          MPI_Bcast(buffer2, count, type, 1, comm); break;
  case 1: MPI_Bcast(buffer2, count, type, 1, comm);
          MPI_Bcast(buffer1, count, type, 0, comm); break;
}
```

- Welcher Fehler würde auftreten, wenn das Laufzeitsystem die jeweils ersten beiden Anweisungen eines Prozesses aufeinander bezieht und als eine gemeinsame Operation betrachtet (Hinweis: Es ist eine Bedingung für die `MPI_Bcast()`-Argumente verletzt)?
- Was passiert, wenn das Laufzeit-System die Kommunikations-Anweisungen mit gleichem Wurzel-Prozess aufeinander bezieht, jedoch keine Systempuffer verwendet werden?

Teilaufgabe (b) (5 Punkte)

Ohne Verwendung von Systempuffern können kollektive Kommunikations-Anweisungen synchronisierend wirken. Betrachten Sie das folgende Programm-Fragment, in dem zweifelhaft ist, welche Nachricht der Prozess Π_1 mit welchem `MPI_Recv()`-Aufruf empfängt:

```
switch (my_rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Send( buf2, count, type, 1, tag, comm); break;
  case 1:
    MPI_Recv( buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Recv( buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status); break;
  case 2:
    MPI_Send( buf2, count, type, 1, tag, comm);
    MPI_Bcast(buf1, count, type, 0, comm); break;
}
```

Für die Kommunikation zwischen den drei Prozessoren können unterschiedliche Ablaufreihenfolgen auftreten (Listing links und rechts):

Π_0	Π_1	Π_2	Π_0	Π_1	Π_2
	MPI_Recv ←	MPI_Send	MPI_Bcast		
MPI_Bcast	MPI_Bcast	MPI_Bcast	MPI_Send →	MPI_Recv	
MPI_Send →	MPI_Recv			MPI_Bcast	
				MPI_Recv ←	MPI_Send
					MPI_Bcast

Welche der Reihenfolgen können mit, welche ohne Verwendung von Systempuffern eintreten (Mit Argumentation)? Welcher Variante kann daher ein nicht-deterministisches Programm generieren?

In dieser Aufgabe sollte das unterschiedliche Verhalten von Programmen in Abhängigkeit des verwendeten Laufzeitsystems betrachtet werden. Die Laufzeitsysteme können dabei völlig verschiedene Verhaltensweisen haben, beispielsweise könnten Nachrichten je nach Länge in einen Systempuffer geschrieben werden oder nicht. Wichtig ist, sich nicht auf ein Verhalten zu verlassen, sondern vom Verhalten unabhängig korrekt arbeitende Programme zu schreiben. Zur Synchronisation verwendet man etwa lieber explizite Anweisungen wie `MPI_Barrier()` *anstatt globale Kommunikations-Anweisungen.*

Übung 15 Einfache Parallelisierung des Jacobi-Verfahrens mit MPI (10 Punkte)

Für lineare Gleichungssysteme $A\mathbf{x} = \mathbf{b}, A \in \mathbb{R}^{n\times n}$, $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$, $n \in \mathbb{N}$, sind direkte Lösungsverfahren für große n meist ineffizient. Daher verwendet man oft iterative Verfahren wie das *Jacobi-Verfahren*. Man erhält es durch Aufspalten der Systemmatrix A in die obere und untere Dreicksmatrix U und L sowie die Diagonalmatrix D, $A = D + L + U$. Dies führt auf die Fixpunktgleichung $\mathbf{x} = D^{-1}(\mathbf{b} - (A - D)\mathbf{x})$, die unter gewissen Vorraussetzungen mit einer Fixpunktiteration gelöst werden kann: $\mathbf{x}^{(m+1)} = D^{-1}(\mathbf{b} - (A - D)\mathbf{x}^{(m)})$, Index $m \in \mathbb{N}$ indiziert den Iterationsschritt. Die i.te Gleichung für den $(m + 1)$. Schritt des Verfahrens lautet dann

$$x_i^{(m+1)} = \frac{b_i - \sum_{j\neq i} a_{ij} x_j^{(m)}}{a_{ii}}.$$

Algorithmisch hat das Jacobi-Verfahren die Form:

```
Wahl eines Startwertes fuer x;
for m=1,2,...
  for i=1,2,...,n
    y_i = 0;
    for j=1,2,...,i-1,i+1,...,n
      y_i = x_i + a_ij * x_j^(m-1);
    y_i = (b_i - x_i)/a_ii;
x^m = y;
Konvergenztest; Abbruch falls Toleranz erreicht.
```

Der Hilfsvektor `y` speichert Zwischenergebnisse. Der Vektor $\mathbf{x}$ darf nicht innerhalb der Iterationen neu bechrieben werden, da seine Werte in jeder Zeile benötigt werden. Die Abbruchbedingung ergibt sich aus der Berechung des Residuums $\mathbf{r}$ im m.ten Schritt, $\mathbf{r}^m := \mathbf{b} - A\mathbf{x}^{(m)}$: Ist eine geeignete Norm des Residuums (z.B. die Maximumsnorm $||\mathbf{r}||_\infty$) kleiner einer gegeben Toleranz $\epsilon \in \mathbb{R}_+$, wird abgebrochen.

Die Berechnungen einer Iteration hängen nur von der vorhergehenden Lösung ab und es gibt es keine Datenabhängigkeiten zwischen den neu zu berechnenden Werten x_i. Daher ist das Jacobi-Verfahren leicht zu parallelisieren.

Auf der Homepage finden Sie eine sequentielle Implementieren des Jacobiverfahrens. Erweitern Sie das Programm zur parallelen Berechnung einer Lösung $\mathbf{x}$. Eine mögliche Strategie ist beispielsweise, A und $\mathbf{b}$ in Querstreifen der Höhe (Anzahl Zeilen) h aufzuteilen (P ist die Anzahl der Prozessoren). Jeder Prozessor erhält in Schritt m eine Kopie der vorhergehenden Lösung $\mathbf{x}^{(m-1)}$, um die neuen Werte x_i seines Streifens zu berechnen. Jeder Iteration muss jeder Prozess die neuen Werte seines Teilbereiches allen anderen Prozessen bekanntmachen, etwa per Multi-Broadcast.

Gerne können Sie andere Strategien implementieren, näheres findes Sie z.B. im Buch *Parallele und verteilte Programmierung* von *Rauber/Rünger*. Messen Sie den Speed-up gegenüber der sequentiellen Version ($P = 1$) für verschiedene Problemgrößen n und Prozessoranzahlen P. Damit das Verfahren konvergieren kann, sollte Ihre Matrix A strikt diagonaldominant sein, d.h. es sollte gelten $\sum_{j=1;j\neq i}^{n} |a_{ij}| < |a_{ii}|$, $\forall i \in \{1, .., n\}$.