MATHEMATISCHES INSTITUT
DER UNIVERSITÄT ZU KÖLN

Dr. Stephanie Friedhoff, Dr. Martin Lanser

Einführung in HPC
Sommersemester 2016

Übung 2

Hinweis: Schreiben Sie bitte jede Aufgabe auf ein neues Blatt und auf **jedes Blatt Ihren Namen**. Auf die erste Seite Ihrer Übung schreiben Sie bitte zusätzlich zu Ihrem Namen Ihre Matrikelnummer.

Aufgabe 1 (4 Punkte).
In dieser Aufgabe gehen wir von **blockierender Kommunikation** aus, d. h. ein Datenaustausch ist erst beendet, wenn sowohl der Sende- als auch der zugehörige Empfangsbefehl ausgeführt sind. Ein **Deadlock** in einem parallelen Algorithmus liegt vor, wenn ein Prozessor auf den Empfang von Daten wartet, die niemals gesendet werden.

Gegeben sei ein Parallelrechner mit Ringtopologie bestehend aus $p > 1$ Prozessoren $P_0, P_1, \ldots, P_{p-1}$. Jeder Prozessor P_i habe Zugriff auf eine Datenmenge d_i, $i = 0, 1, \ldots, p-1$. Entwerfen Sie einen parallelen Algorithmus ohne Deadlocks um einen Links-Shift der Daten durchzuführen, d. h. nach Abschluss des Links-Shifts hat jeder Prozessor P_i zusätzlich Zugriff auf die Datenmenge $d_{(i+1) \mod p}$.

Aufgabe 2 (2 + 2 Punkte).
Gegeben sei ein Parallelrechner mit Hypercube-Topologie bestehend aus 2^d Prozessoren.

(a) Bei einer **broadcast** Operation sendet ein Prozessor eine Datenmenge D an alle anderen Prozessoren.

Entwerfen Sie einen parallelen Algorithmus ohne Deadlocks zur Durchführung einer broadcast-Operation zum Versenden einer Datenmenge D von Prozessor P_b, $0 \leq b \leq 2^d - 1$ zu allen anderen Prozessoren.

(b) Eine **reduce-to-all** Operation ist wie eine normale Reduktionsoperation (fan-in), bei der allerdings nach Abschluss der Operation alle Prozessoren (anstelle von nur einem Prozessor) Zugriff auf das Ergebnis haben. Diese Operation könnte als normale Reduktionsoperation mit anschließender broadcast-Operation implementiert werden. Allerdings würde dieses Vorgehen die doppelte Zeit in Anspruch nehmen.

Wir nehmen an, dass ein bidirektionaler Datenaustausch zwischen zwei benachbarten Prozessoren die gleiche Zeit in Anspruch nimmt, wie das (unidirektionale) Versenden von Daten (inklusive der zugehörigen Empfangsoperation) von einem Prozessor zu einem benachbarten Prozessor. Entwerfen Sie einen parallelen Algorithmus ohne Deadlocks zur Durchführung einer reduce-to-all Operation in der *gleichen* Zeit, die eine normale Reduktionsoperation in Anspruch nehmen würde, d. h. verwenden Sie d parallele (bidirektionale) Kommunikationsoperationen und d parallele Additionen.

Aufgabe 3 (0 Punkte).

(a) Installieren Sie eine MPI-Implementierung. Frei verfügbare MPI-Implementierungen sind z. B.:

(i) **MPICH / MPICH2** (http://www.mpich.org)

(ii) **Open MPI** (http://www.open-mpi.org)

(b) Übersetzen Sie das folgende MPI-Programm (`mpi_hello.c`; auf der Homepage zur Vorlesung als Download verfügbar) und führen Sie es auf 1, 2 sowie 4 Prozessen aus.

Listing 1: MPI Hello World

```c
#include <stdio.h>
#include "mpi.h"

int main (int argc, char **argv)
{
    int err;
    int size, rank;

    /* Initialisiere MPI-Umgebung */
    err = MPI_Init (&argc, &argv);

    /* Bestimme meinen Rang im Kommunikator MPI_COMM_WORLD sowie
       die Anzahl der Prozesse im Kommunikator MPI_COMM_WORLD. */
    err = MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    err = MPI_Comm_size (MPI_COMM_WORLD, &size);

    /* Ausgabe von Rang und Groesse auf jedem Prozess */
    printf("Hello world von Prozess %d von %d.\n", rank, size);

    /* Beende MPI-Umgebung */
    err = MPI_Finalize ();

    return 0;
}
```

Hinweis: MPI-Programme können mittels `mpicc` übersetzt werden. Mit den Befehlen `mpiexec -np <Anzahl Prozesse>` bzw. `mpirun -n <Anzahl Prozesse>` können Sie Ihr Programm ausführen, z. B.

```
mpicc mpi_hello.c -o mpi_hello
mpirun -n 4 ./mpi_hello
Hello world von Prozess 1 von 4.
Hello world von Prozess 2 von 4.
Hello world von Prozess 3 von 4.
Hello world von Prozess 0 von 4.
```

Programmieraufgabe 1 (10 Punkte).

Implementieren Sie ein paralleles Programm mit $2^N,\ N \in \mathbb{N}$ MPI Prozessen (*rank 0* bis *rank* $2^N - 1$), das alle Einträge v_i eines Vektors v der Länge $n * 2^N,\ n \in \mathbb{N}$ aufsummiert. Jeder MPI Prozess sollte zunächst einen Teil / Block des Vektors der Länge n bearbeiten, bevor die resultierenden Teilsummen kommuniziert und addiert werden. Das Gesamtergebnis sollte am Ende des Programms auf *rank 0* ausgegeben werden. Implementieren Sie die nötige Kommunikation

a) mithilfe der Fan-in Strategie und Punkt-zu-Punkt Kommunikationen,

b) mithilfe einer globalen Kommunikation.

Messen Sie für beide Fälle a) und b) die benötigte Kommunikationszeit und die gesamte Programmlaufzeit für 2, 4, 8, 16 und 64 MPI Prozesse und verschiedene n. Stellen Sie die Ergebnisse tabellarisch oder grafisch dar. Was fällt Ihnen auf?

Hinweise:

- Allokieren Sie für den lokalen Block des Vektors v der Länge n auf *rank i* (Einträge v_{i*n} bis $v_{(i+1)*n-1}$) ein **array** der Länge n auf jedem MPI Prozess. Ein globaler Vektor v muss nicht explizit implementiert werden, da jeder MPI Prozess nur Zugriff auf besagten lokalen Block der Länge n benötigt.
- Wählen Sie $(i + 1)/n$ für alle Einträge des arrays auf *rank i*. Dann können Sie das Endergebnis Ihrer Summe mit der Gaußschen Summenformel überprüfen!
- Vermeiden Sie **Deadlocks** in Ihrer Fan-in Implementierung!
- Ihr Programm muss für Prozesszahlen $p! = 2^N$ nicht lauffähig sein.
- Nutzen Sie für Ihre Fan-in Implementierung die Befehle **MPI_Send** und **MPI_Recv**.
- Schauen Sie sich den Befehl **MPI_Allreduce** oder **MPI_Reduce** für die globale Kommunikation an.
- Für die Zeitmessungen eignet sich der Befehl **MPI_Wtime**.

Abgabedatum: 09. Mai 2016 bis 12:00 Uhr im entsprechenden Kasten in Raum 3.01 des Mathematischen Instituts oder am Ende der Vorlesung.