Karlsruher Institut für Technologie

KASTEL / IPD

Prof. Dr. Sebastian Erdweg erdweg@kit.edu
Prof. Dr. Ralf H. Reussner reussner@kit.edu

Programmierparadigmen – WS 2025/26

https://dsis.kastel.kit.edu/1181_1197.php

Blatt 1: C/C++, MPI Besprechung: 19.12.2024

1 Grundlagen der Parallelprogrammierung: Daten- und Taskparallelismus

Gegeben seien folgende Szenarien

1. Es sollen Kanten in einem Bild (angegeben durch eine Matrix bestehend aus Pixeln) detektiert werden. Hierfür können Faltungsmatrizen zur Kantendetektion angewandt werden, welche für jeden Pixel des Bildes einen Farbgradienten berechnen. Eine Faltungsmatrix beschreibt hierbei für einen Pixel, welche der umliegenden Pixel für die Berechnung des Gradienten verwendet werden und den gewichteten Einfluss jedes umliegenden Pixels auf den Gradienten.
2. Es soll ein Integral numerisch berechnet werden.
3. Es soll ein Webserver programmiert werden, der mehrere Anfragen bearbeiten kann.

Aufgabe: Nennen Sie für jedes Szenario, ob Daten- oder Task-Parallelismus zur Beschleunigung verwendet werden kann und begründen Sie Ihre Entscheidung. Beschreiben Sie zusätzlich für jedes Szenario, wie es durch Parallelisierung beschleunigt werden könnte. Sie brauchen hierfür keine konkreten Parallelisierungskonstrukte zu nennen.

2 C: Zeiger-Arithmetik, Arrays

Welche Ausgabe erzeugt das folgende C-Programm? Begründen Sie Ihre Antwort kurz und machen Sie Ihren Lösungsweg deutlich, indem Sie Ihre Auswertung jeweils unter die Programmzeilen schreiben.

Hinweis: `printf("%i",`i`)` gibt den Zahlenwert eines Integers i aus.

```
#include <stdio.h>

int global[] = {1, 2, 3, 4, 5};

int *magic(int x[], int y) {
        printf("m");
        global[1] = *(global + y) + 3;
        return &x[y - 2];
}

int main() {
        printf("%i", *magic(&global[1], *(global + 1)));
        return 0;
}
```

3 MPI: Send/Receive und deren Tücken

Eine Firma entwickelte folgendes MPI-Programm. Es ist dafür vorgesehen, von zwei MPI-Prozessen ausgeführt zu werden. Den Quellcode des Programms finden Sie im ILIAS-Kurs. Eine kurze Wiederholung zur Installation und Ausführung von MPI finden Sie am Ende dieses Übungsblatts.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "mpi.h"
5
6  int main(int argc, char* argv[]) {
7    MPI_Init(&argc, &argv);
8
9    int my_rank;
10   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
11
12   if (my_rank < 2) {
13     int        other_rank = 1 - my_rank;
14     int        tag        = 0;
15     char       message[14];
16     sprintf(message, "Hello, I am %d", my_rank);
17     MPI_Status status;
18
19     MPI_Send(message, strlen(message) + 1, MPI_CHAR, other_rank,
20              tag, MPI_COMM_WORLD);
21     MPI_Recv(message, 100, MPI_CHAR, other_rank,
22              tag, MPI_COMM_WORLD, &status);
23     printf("%s\n", message);
24   }
25
26   MPI_Finalize();
27   return EXIT_SUCCESS;
28 }
```

Während des Testens durch die Firma verhielt sich das Programm wie erwartet. Nachdem es jedoch an Kunden ausgeliefert wurde, beschwerten sich diese prompt darüber, dass das Programm hängenbleibt.

1. Wieso bleibt das Programm hängen?
2. Verändern Sie das Programm so, dass es auf beliebigen MPI-Umgebungen korrekt ausgeführt werden kann. Versuchen Sie so viele verschiedene Lösungen zu finden wie möglich.

4 MPI: Von Send/Receive zu kollektiven Operationen

Gegeben sei die untenstehende Methode, welche in einem MPI-Programm mit einer beliebigen Anzahl an Prozessen aufgerufen wird. Sie können davon ausgehen, dass die Initialisierung und Finalisierung von MPI vom Aufrufer übernommen wird. Weiterhin können Sie davon ausgehen, dass `elementCount` immer ein Vielfaches der Anzahl verfügbarer Prozesse ist und per Präprozessor definiert ist.

```
1  void operate(int elementCount, int elements[elementCount]) {
2      int rank;
3      int processCount;
4      int rootRank = 0;
5
6      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7      MPI_Comm_size(MPI_COMM_WORLD, &processCount);
8
9      int elementsPerProcess = elementCount / processCount;
10     int local[elementsPerProcess];
11     MPI_Scatter(elements, elementsPerProcess, MPI_INT, local,
12         elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);
13
14     int offset = 0;
15     if (rank == rootRank) {
16         for (int i = 0; i < elementCount; i++) {
17             if (elements[i] < offset) offset = elements[i];
18         }
19     }
20
21     if (rank == rootRank) {
22         for (int process = 0; process < processCount; process++) {
23             if (process != rootRank) {
24                 MPI_Send(&offset, 1, MPI_INT, process, 0,
                       MPI_COMM_WORLD);
25             }
26         }
27     } else {
28         MPI_Recv(&offset, 1, MPI_INT, rootRank, 0, MPI_COMM_WORLD, NULL);
29     }
30
31     for (int i = 0; i < elementsPerProcess; i++) {
32         local[i] = local[i] - offset;
33     }
34
35     MPI_Gather(local, elementsPerProcess, MPI_INT, elements,
36         elementsPerProcess, MPI_INT, rootRank, MPI_COMM_WORLD);
37
38     if (rank == rootRank) {
39         for (int i = 0; i < elementCount; i++) {
40             printf("%i␣", elements[i]);
41         }
42     }
43 }
```

1. Erklären Sie, welche Funktion das Programm erfüllt und geben Sie die Ausgabe für folgende Eingabe an: `(4, [-1, 2, 9, 5])` [3 Punkte]

2. Schreiben Sie einen Ersatz für die Zeilen 21 bis 29, der ausschließlich aus dem Aufruf einer MPI-Operation besteht. [2 Punkte]

3. Die Zeilen 15 bis 19 werden rein sequentiell ausgeführt. Erläutern Sie die notwendigen Schritte, um diese Zeilen mittels MPI möglichst effizient parallelisieren können. Geben Sie den notwendigen Programmcode als Ersatz für die Zeilen an. [3 Punkte]

5 MPI: Reduce und der Weg zurück zu Send/Receive

1. Analysieren Sie folgenden Ausschnitt aus einem MPI-Programm unter der Annahme, dass es mit 4 Prozessen ausgeführt wird. Geben Sie in untenstehender Tabelle an, welche Werte die Puffer `sendbuffer` und `recvbuffer` nach Ausführung von `MPI_Reduce` innerhalb der jeweiligen Prozesse enthalten.

```
1  int size, rank;
2  MPI_Comm_size(MPI_COMM_WORLD, &size);
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4
5  int sendbuffer[4];
6  int recvbuffer[4];
7
8  for (int i = 0; i < 4; i++) {
9      sendbuffer[i] = rank + 2*i;
10 }
11
12 MPI_Reduce(sendbuffer, recvbuffer, 4, MPI_INT,
13             MPI_SUM, 0, MPI_COMM_WORLD);
```

`sendbuffer` bei Rank 0				
`sendbuffer` bei Rank 1				
`sendbuffer` bei Rank 2				
`sendbuffer` bei Rank 3				
`recvbuffer` bei Rank 0				

2. Implementieren Sie die kollektive Operation `MPI_Reduce` für das Aufsummieren von `int`-Arrays mithilfe der folgenden MPI-Funktionen:
   `MPI_Send`, `MPI_Recv`, `MPI_Comm_size` und `MPI_Comm_rank`.

   Ergänzen Sie dazu den unten angegebenen Funktionsheader `my_int_sum_reduce` so, dass ein Aufruf der Funktion die Daten in derselben Weise verteilt, wie ein Aufruf von `MPI_Reduce` mit dem Parameterwert `MPI_SUM`.

   Hinweis: Sie dürfen davon ausgehen, dass `my_int_sum_reduce` nur mit gültigen Argumenten aufgerufen wird. Sie brauchen sich also nicht um Fehlerbehandlung aufgrund falscher Argumente zu kümmern.
   Vermeiden Sie Aufrufe von `MPI_Send`, bei denen Sender und Empfänger identisch sind, da dies zu einem Deadlock führen kann.

   Verwenden Sie für Ihre Methode die folgende Signatur:

```
1 void my_int_sum_reduce(int *sendbuf, int *recvbuf, int count,
2                        int root, MPI_Comm comm)
```

6 MPI: Matrizenmultiplikation [alte Klausuraufgabe, 11.5 Punkte]

Gegeben seien zwei $n \times n$-Matrizen s A und B als zweidimensionale Integer-Array. Die multiplikative Verknüpfung dieser zwei Matrizen sei wie folgt definiert:

$$C = A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = (c_{ij}), \tag{1}$$

wobei gilt:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{2}$$

1. Diese Matrizenmultiplikation soll mithilfe von MPI auf genau *procs* vielen Knoten des Standard-Kommunikators `MPI_COMM_WORLD` verteilt werden. Hierbei sollen Sie untenstehenden Methode vervollständigen, welche von einer `main`-Methode zur Berechnung aufgerufen wird. Achten Sie bei Ihrer Implementierung darauf, **wenn möglich**, nur für die Berechnung **relevante Teile** der Matrizen zu übertragen. Benutzen Sie hierfür geeignete kollektive Operationen. Geben Sie für diese die **gesamte Parameterbelegung** an. Die Einträge sollen am Ende mit einer einzigen kollektiven Operation auf **Knoten 0 in Ergebnismatrix c** zusammengeführt werden.

   Gehen Sie davon aus, dass folgende Annahmen gelten:

   - Für `n` gilt `n mod procs` $= 0$.
   - Die `main`-Methode nimmt alle nötigen Initalisierungen und Finalisierungen der MPI-Umgebung vor.
   - Die Daten liegen beim Aufruf von `matrixMultiply` nur auf dem Rootprozess mit dem `rank 0` vor. Auf den anderen Prozessen sind die Arrays zwar allokiert, aber nicht initialisiert.
   - Die Arrays `a`, `b` und `c` sind statisch allokiert. Beachten Sie hierbei, dass in C statisch allokierte zweidimensionale Arrays im Speicher sequentiell repräsentiert sind. Eine schematische Repräsentation für `a[n][n]` mit `n = 3` im Speicher ist:

...	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]	...

   Vervollständigen Sie die matrixMultiply Methode:

```
1  /* n sei die Anzahl der Zeilen und Spalten,
2   * beliebig per Präprozessor vordefiniert */
3
4  void matrixMultiply(int a[n][n], int b[n][n], int c[n][n])
5  {
6      int procs;
7      int rank;
8      MPI_Comm_size(MPI_COMM_WORLD, &procs);
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11     // hier ergänzen
12
13 }
```

2. Die oben erstellte Lösung setzt voraus, dass die Anzahl der Zeilen und Spalten der Matrizen ein Vielfaches der Anzahl Prozessoren des Systems sei. Woraus ergibt sich diese Beschränkung? Erklären Sie stichpunktartig, wie sich diese Beschränkung lösen ließe.

3. *Zusatzaufgabe (nicht Teil der ursprünglichen Klausuraufgabe):*
   Versuchen Sie Ihre Lösung weiter zu optimieren. Ab welcher Matrizengröße kann man einen Speedup beobachten?

Appendix: MPI Installieren und Ausführen

Sie können Open MPI über den Paket-Manager Ihres Betriebssystems oder über die Open-MPI-Webseite[1] installieren.

```
# Für Ubuntu:
sudo apt install openmpi-bin openmpi-doc libopenmpi-dev
```

Wenn Sie MPI in einem C-Programm benutzen möchten, importieren Sie die Bibliothek mit dem folgenden Befehl am Anfang Ihrer C-Datei.

```
#include "mpi.h"
```

Die C-Dateien (z.B. `mein-programm.c`) können Sie mit dem MPI-Compiler `mpicc` kompilieren lassen. Intern wird dabei der konfigurierte C-Compiler benutzt (z.B. gcc). Sie können dem Befehl auch mehrere Dateipfade übergeben. Mit dem Parameter `-o` geben Sie den Namen der Ausgabedatei an.

```
mpicc mein-programm.c -o mein-programm.out
```

Die erzeugte Datei (z.B. `mein-programm.out`) können Sie mit dem Befehl `mpirun` ausführen. Dabei übergeben Sie mit dem Parameter `-np` die Anzahl der Nodes, auf denen das (gleiche) Programm ausgeführt werden soll. Am Ende des Befehl können Sie optional zusätzliche Argumente für das Programm angeben.

```
mpirun -np 4 mein-programm.out 73 pp
```

[1] https://www.open-mpi.org/