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.

**Beispiellösung**:

- Für die Kantendetektion (erstes Szenario) kann Datenparallelismus angewandt werden. Das Bild kann als eine Menge von Daten (Pixeln) angesehen werden, welche in mehrere Blöcke aufgeteilt werden können. Für jeden Block führt jeweils ein Thread die Berechnung des Farbgradienten für jeden Pixel darin durch. Hierfür werden dem Thread die Pixel im Block und weitere für die Filterung benötigten Pixel übergeben. Jeder Thread wendet dann auf die ihm zugeteilten Pixel die Faltungsmatrix an, um den Farbgradienten zu bestimmen. Alle gefilterten Werte (alle Farbgradienten) der Blöcke werden dann wieder zu dem gefilterten Bild zusammengefügt.
- Für die Berechnung des Integrals (zweites Szenario) kann ebenfalls Datenparallelismus angewandt werden. Hierbei wird die Kurve (die Daten) in Teilabschnitte unterteilt. Für jeden Teilabschnitt wird von einem Thread das Integral berechnet und anschließend werden alle Ergebnisse zusammengefügt. Somit führt jeder Thread die gleiche Aufgabe auf unterschiedlichen Daten aus.
- Ein Webserver (drittes Szenario) kann durch Task-Parallelismus beschleunigt werden, indem jede Anfrage einem Thread zugewiesen wird. Jeder Thread einer Webserver-Instanz kann hierbei verschiedene Aufgaben, wie z.B. Login-Anfragen oder Abfragen von Datensätzen, bearbeiten. Somit kann jeder Thread seine Aufgabe unabhängig von den anderen Threads ausführen.

## 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;
}
```

**Beispiellösung**:

*Ausgabe:* `m6`

| Methode | Operationen |
|---|---|
| `main` | • `&global[1]` liefert die Adresse von `global` ab Indexposition 1 |
| | • `*(global+1)` liefert den Wert von `global` an Indexposition 1, also 2 |
| | • Main ruft die Methode `magic` auf mit den Parametern `x = [2,3,4,5]` (d.h. alias von `global` ab Indexposition 1) und `y = 2` (d.h. Wert von `global` an Indexposition 1). |
| `magic` | • Zunächst Ausgabe von "m“ |
| | • `*(global+y)` ergibt dann `*(global+2)`, d.h. Wert von `global` an Indexposition 2, also 3 |
| | • Danach wird `global` an Position 1 aktualisiert mit 3+3 = 6; d.h. `global = [1,6,3,4,5]`. |
| | • Mit `&x[y-2]` → `&x[2-2]` → `&x[0]` wird die Adresse des ersten Elements von `x` (=6) zurückgegeben. |
| `main` | `printf` dereferenziert die übergebene Adresse und gibt den Wert 6 aus. |

## 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.

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

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int my_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank < 2) {
    int        other_rank = 1 - my_rank;
    int        tag        = 0;
    char       message[14];
    sprintf(message, "Hello, I am %d", my_rank);
    MPI_Status status;

    MPI_Send(message, strlen(message) + 1, MPI_CHAR, other_rank,
             tag, MPI_COMM_WORLD);
    MPI_Recv(message, 100, MPI_CHAR, other_rank,
             tag, MPI_COMM_WORLD, &status);
    printf("%s\n", message);
  }

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

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.

**Beispiellösung**:

*Grund für das Hängenbleiben:*

Das Programm hängt sich bei den Kunden auf, da diese eine MPI-Umgebung verwenden, welche im Standardmodus nicht puffert oder in welcher der Puffer nicht ausreichend groß ist, um die Nachricht zu puffern. Die Firma verwendete zum Testen eine Umgebung, welche im Standardmodus puffert. Deshalb lief das Programm dort problemlos.

Im Programm senden sich die Prozesse gegenseitig Nachrichten und benutzen dabei die blockierende Sendeoperation im Standardmodus. Diese Operation blockiert, bis der Sendepuffer (hier: `message`) gefahrlos wiederverwendet werden kann. Wenn die MPI-Umgebung im Standardmodus den Systempuffer verwendet (und in diesem ausreichen Platz für `message` ist) kann der Aufruf der Sendeoperation zurückkehren und der Prozess weiterarbeiten, bevor der Empfänger die Empfangsoperation aufgerufen hat. Wenn die MPI-Umgebung den Systempuffer im Standardmodus nicht verwendet, wird der Sender blockiert bis der Empfänger die Empfangsoperation aufgerufen hat.

Das Programm bleibt hängen, da beide Prozesse beim Senden blockieren und darauf warten, dass der jeweils andere die Empfangsoperation aufruft (was nie passiert).

*Lösungsmöglichkeiten (Beispiele):*

- *Unterschiedliches Verhalten (blockierenden Standardmodus beibehalten):* Implementiere unterschiedliche Verhalten für die Prozesse, so dass die Reihenfolge der Sende- und Empfangsoperationen unterschiedlich ist.
  Z.B.: Prozess 0 ruft die Operationen in folgender Reihenfolge auf: `MPI_Send` → `MPI_Recv`. Prozess 1 ruft die Operationen hingegen folgendermaßen auf: `MPI_Recv` → `MPI_Send`. Die Operationen blockieren zwar immer noch, aber Prozess 0 kann weiterarbeiten, sobald Prozess 1 empfangen hat. Dies entspricht einer Standardlösung um Deadlocks zu vermeiden (Bedingung *circular wait* aushebeln).
- *Verwende den gepufferten Modus:* Ersetze `MPI_Send` durch `MPI_Bsend` und halte Systempuffer bereit, die groß genug sind um `message` aufzunehmen. Dies kann man mit `MPI_Buffer_attach` erreichen. Dies versichert, dass die MPI-Umgebung die gesendeten Daten im Systempuffer speichert und der Sender gleich fortfahren kann. Somit müssen Prozess 0 und 1 nicht aufeinander warten.
- *Verwende nicht blockierende Kommunikation (Standardmodus beibehalten):* Ersetze `MPI_Send` durch `MPI_Isend` und verwende einen weiteren Puffer für `MPI_Recv`. Dadurch ruft jeder Prozess eine nicht-blockierende Sendeoperation auf.
- *Verwende eine kombinierte Sende- und Empfangsoperation:* Ersetze die Aufrufe von `MPI_Send` und `MPI_Recv` durch einen Aufruf von `MPI_Sendrecv`. Dieser benötigt allerdings zwei Puffer (einen zum Senden und einen zum Empfangen). Alternativ kann man auch `MPI_Sendrecv_replace` verwenden. Dadurch braucht man nur einen Puffer.

Natürlich gibt es auch noch weitere Lösungen, welche teils die Lösungen von oben kombinieren.

# 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]

   **Beispiellösung**: Das Programm erhöht alle Werte des Eingabe-Array `elements` um die Differenz zwischen null und dem kleinsten Wert des Arrays, falls dieser kleiner als null ist. Ansonsten tut das Programm nichts.

   Ausgabe: ”0 3 10 6“

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

   **Beispiellösung**:

   ```
   MPI_Bcast(&offset, 1, MPI_INT, rootRank, MPI_COMM_WORLD);
   ```

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]

   **Beispiellösung**:

   Es kann lokal in jedem Prozess für das Teilarray der Minimalwert berechnet werden und dann per `MPI_Reduce` das globale Minimum über diese lokalen Minima ermittelt werden.

   ```
   int localOffset = 0;
   for (int i = 0; i < elementsPerProcess; i++) {
       if (local[i] < localOffset) localOffset = local[i];
   }
   MPI_Reduce(&localOffset, &offset, 1, MPI_INT, MPI_MIN,
       rootRank, MPI_COMM_WORLD);
   ```

# 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 | 0 | 2 | 4 | 6 |
| `sendbuffer` bei Rank 1 | 1 | 3 | 5 | 7 |
| `sendbuffer` bei Rank 2 | 2 | 4 | 6 | 8 |
| `sendbuffer` bei Rank 3 | 3 | 5 | 7 | 9 |
| `recvbuffer` bei Rank 0 | 6 | 14 | 22 | 30 |

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)
```

**Beispiellösung**:

```
1  void my_int_sum_reduce(int *sendbuf, int *recvbuf, int count,
2                         int root, MPI_Comm comm) {
3      int size, rank;
4      MPI_Comm_size(comm, &size);
5      MPI_Comm_rank(comm, &rank);
6
7      if (rank == root) {
8          /* Initialize recvbuf with our own values. */
9          for (int i = 0; i < count; ++i)
10             recvbuf[i] = sendbuf[i];
11
12         /* Receive values from every other node and accumulate. */
13         for (int i = 0; i < size; ++i) {
14             if (i == root)
15                 continue;
16
17             int other[count];
18             MPI_Recv(other, count, MPI_INT, i, 0, comm,
                   MPI_STATUS_IGNORE);
19             for (int j = 0; j < count; ++j)
20                 recvbuf[j] += other[j];
21         }
22     } else {
23         /* Send our values to root. */
24         MPI_Send(sendbuf, count, MPI_INT, root, 0, comm);
25     }
26 }
```

## 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 }
```

**Beispiellösung**:

```
1  void matrixMultiply( int a[n][n], int b[n][n], int c[n][n])
2  {
3      int procs;
4      int rank;
5      MPI_Comm_size(MPI_COMM_WORLD, &procs);
6      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7
8      int from = rank * n/procs;
9      int to = (rank+1) * n/procs;
10     MPI_Bcast(b, n*n, MPI_INT, 0, MPI_COMM_WORLD);
11     MPI_Scatter(a, n*n/procs, MPI_INT, a[from], n*n/procs,
12                 MPI_INT, 0, MPI_COMM_WORLD);
13     int i,j,k;
14     for (i = from; i < to; ++i) {
15         for (j = 0; j < n; ++j) {
16             c[i][j] = 0;
17             for (k = 0; k < n; ++k) {
18                 c[i][j] += a[i][k] * b[k][j];
19             }
20         }
21     }
22
23     MPI_Gather(c[from], n*n/procs, MPI_INT, c, n*n/procs,
24                MPI_INT, 0, MPI_COMM_WORLD);
25 }
```

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.

   **Beispiellösung**: Die Beschränkung ergibt sich daraus, dass die kollektiven Operationen `MPI_Gather` und `MPI_Scatter` an jeden beteiligten Prozess die gleiche Anzahl Daten verteilen (scatter), bzw. die gleiche Menge Daten holen (gather). Variiert die Menge der Daten je Prozess, so können beispielsweise die Vektoroperationen `MPI_Gatherv` bzw. `MPI_Scatterv` eingesetzt werden, die es erlauben eine für jeden Prozess individuelle Datenmenge zu wählen.

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