Karlsruhe Institute of Technology

# Programmierparadigmen

## Übung 1 (Reussner) – 19.12.2024

Lars König, M.Sc.

# Organisatorisches

- Übungen zum Vorlesungsteil von Prof. Reussner
  - *Heute (19.12. 14:00)*
  - 09.01. 14:00
  - 16.01. 14:00

- Nächstes Übungsblatt zu Parallelprogrammierung in Java
  - Veröffentlichung: 19.12.
  - Besprechung: 09.01.

# Übersicht Vorlesung

**Programmieren SWT**

Imperativ
Objekt-orientiert

**Programmierparadigmen**

Funktionale Programmierung

Parallelprogrammierung

Design by Contract

(Funktionale Programmierung)

# Task- und Datenparallelismus

Task-Parallelismus

- Aufteilen der Funktion in mehrere Tasks
- Tasks sollten so unabhängig wie möglich sein

Daten-Parallelismus

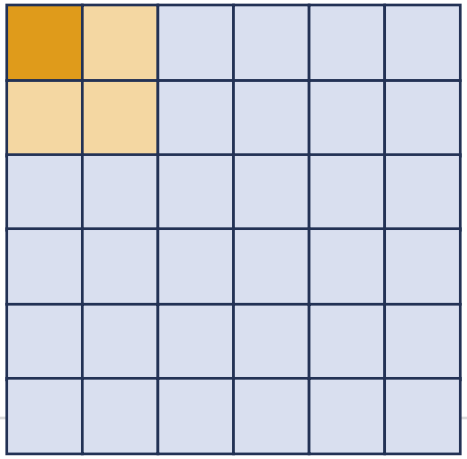
- Aufteilen der Daten auf denen die gleichen Operationen ausgeführt werden
- Tasks sollten soweit möglich nicht auf die gleichen Daten zugreifen

# Aufgabe 1

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.

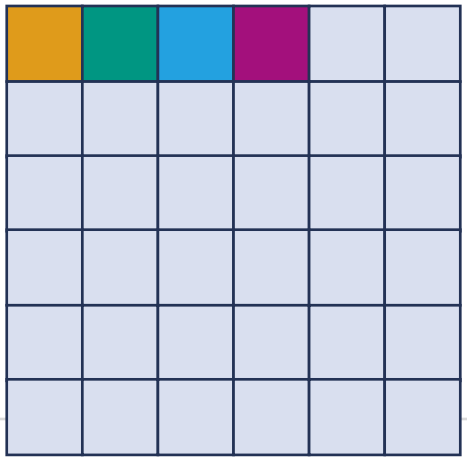
# Aufgabe 1

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.



# Aufgabe 1

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.



# Aufgabe 1

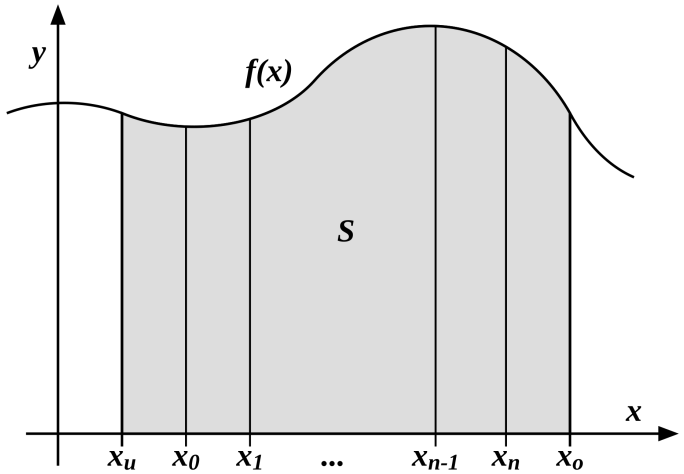
2. Es soll ein Integral numerisch berechnet werden.

3. Es soll ein Webserver programmiert werden, der mehrere Anfragen bearbeiten kann.

# Aufgabe 1

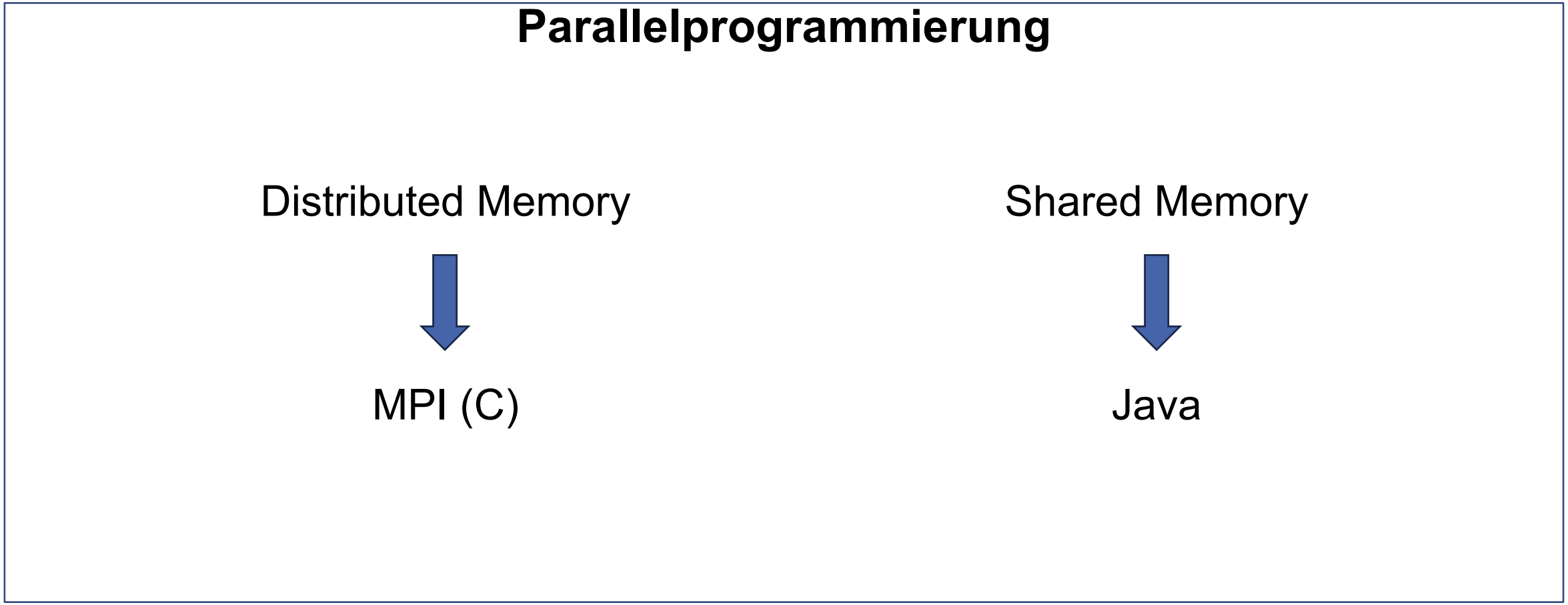
2. Es soll ein Integral numerisch berechnet werden.

3. Es soll ein Webserver programmiert werden, der mehrere Anfragen bearbeiten kann.




```
GET /albums/{id}
GET /artists/{id}
PUT /me/player/play
GET /tracks/{id}
```

# Übersicht Vorlesung



# Pingo

Welchen Typ muss die Variable `c` haben?

```
int a[] = { 1, 3, 5 };
int *b = a;
c = { a, b };
```

# Pingo

Welchen Typ muss die Variable `c` haben?

Lösung:

```
int a[] = { 1, 3, 5 };
int *b = a;
int *c[] = { a, b };
```

# Pingo

Welches Ergebnis wird ausgegeben?

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

# Pingo

Lösung:

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| a654b132 | | 5 | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| c76a593f | | a654b12a | { 1, 3, 5 } |
| 04cfa572 | c | c76a593b | |

# Pingo

Lösung:

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| a654b132 | | 5 | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| c76a593f | | a654b12a | { 1, 3, 5 } |
| 04cfa572 | **c** | **c76a593b** | |

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

c76a593f

+1 auf einem Pointer erhöht den Wert um die Größe des Datentypes

# Pingo

Lösung:

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| a654b132 | | 5 | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| **c76a593f** | | **a654b12a** | { 1, 3, 5 } |
| 04cfa572 | c | c76a593b | |

a654b12a

# Pingo

Lösung:

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| a654b132 | | 5 | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| **c76a593f** | | **a654b12a** | { 1, 3, 5 } |
| 04cfa572 | c | c76a593b | |

a654b132

# Pingo

Lösung:

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| **a654b132** | | **5** | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| c76a593f | | a654b12a | { 1, 3, 5 } |
| 04cfa572 | c | c76a593b | |

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

5

# Pingo

Lösung:

```
int a[] = { 1, 3, 5 };
int *b = a;
c = {a, b};

printf("Result: %d\n", (*(*(c + 1) + 2) + 3));
```

8

| Adresse | Variable | Inhalt | zeigt auf |
|---|---|---|---|
| a654b12a | | 1 | |
| a654b12e | | 3 | |
| **a654b132** | | **5** | |
| 04cfa568 | a | a654b12a | { 1, 3, 5 } |
| 04cfa56c | b | a654b12a | { 1, 3, 5 } |
| c76a593b | | a654b12a | { 1, 3, 5 } |
| c76a593f | | a654b12a | { 1, 3, 5 } |
| 04cfa572 | c | c76a593b | |

# Aufgabe 2

*Selbst mit C-Compiler ausprobieren*

# Message Passing Interface (MPI)

- Standard für Nachrichtenaustausch zwischen verteilten Computersystemen

- **Distributed Memory**, Kommunikation nur mit Nachrichten
- **SIMD**, das gleiche Programm wird auf allen Knoten ausgeführt
  - **MIMD** mit Fallunterscheidung über die Knoten-ID möglich

- Direkte und kollektive Kommunikationsoperationen
- Unterschiedliche Kommunikations- und Aufrufmodi

# Hello World

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

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

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello World! This is node %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

# Hello World

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** args) {
    int rank, size;

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello World! This is node %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

MPI import

# Hello World

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

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

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello World! This is node %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Initialisierung

# Hello World

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

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

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);     Anzahl Nodes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);     ID dieser Node

    printf("Hello World! This is node %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

# Hello World

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

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

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello World! This is node %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Cleanup am Ende

# Aufgabe 3

*Siehe Übungsblatt.*

# Kommunikationsmodi

- Synchronous
  - kein Buffer, beide Seiten warten aufeinander
- Buffered
  - Buffer, keiner muss auf den anderen warten
- Ready
  - kein Buffer, keine Synchronisation, Empfänger wartet auf Sender
- Standard
  - *Synchronous* oder *Buffered*

https://stackoverflow.com/a/47041382

# Aufrufmodi

- Blocking
  - MPI-Funktionen warten bis die Operation abgeschlossen ist
  - Buffer kann nach dem Funktionsaufruf benutzt werden
- Non-Blocking
  - MPI-Funktionen starten die Operation, warten aber nicht auf die Ausführung
  - auf den Buffer wird auch nach dem Funktionsaufruf noch zugegriffen
  - Pointer auf Request-Status, um Fortschritt zu überprüfen

# Direkte Operationen

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buffer, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- `buffer`: Speicherbereich für die Daten
- `count`: Anzahl Elemente
- `datatype`: Typ der Elemente (z.B. `MPI_INT`)
- `dest`: Rank der Ziel-Node
- `source`: Rank der Quell-Node (oder `MPI_ANY_SOURCE`)
- `tag`: Nachrichten-Kontext/-ID (oder `MPI_ANY_TAG`)
- `comm`: Kommunikator
- `status`: stellt Informationen zur Nachricht bereit

# Pingo

- Wie müssen die Ein-, bzw. Ausgabepuffer in den MPI-Aufrufen angegeben werden?

```
int *send_buf = malloc(8 * sizeof(int));
MPI_Send(?, 8, MPI_INT, rank_dest, tag,
     MPI_COMM_WORLD);

int recv_buf = 0;
MPI_Recv(?, 1, MPI_INT, rank_orig, tag,
     MPI_COMM_WORLD, &status);
```

# Pingo

Lösung:

```
int *send_buf = malloc(8 * sizeof(int));
MPI_Send(send_buf, 8, MPI_INT, rank_dest, tag,
     MPI_COMM_WORLD);

int recv_buf = 0;
MPI_Recv(&recv_buf, 1, MPI_INT, rank_orig, tag,
     MPI_COMM_WORLD, &status);
```

# Kollektive Operationen

Beachten Sie: der MPI-Befehl einer kollektiven Operation wird von allen teilnehmenden Nodes aufgerufen

- Synchronize `MPI_Barrier`
  - wartet auf alle teilnehmenden Nodes
- Broadcast `MPI_Bcast`
  - sendet Daten von einer root-Node zu allen Nodes
- Scatter `MPI_Scatter`
  - verteilt Daten von einer root-Node gleichmäßig auf alle Nodes
- Gather `MPI_Gather`
  - sammelt Daten von allen Nodes ein und stellt sie der root-Node zur Verfügung
  - funktioniert invers zu `MPI_Scatter`

# Frage

- Welcher Wert muss für die Parameter “`sendcount`”/”`recvcount`” angegeben werden, wenn alle Werte in “`all_data`” auf $N = 3$ Knoten verteilt werden sollen?

```
int all_data[] = { 2, 3, 5, 7, 11, 13 }; // on root
int *data = malloc(? * sizeof(int));
MPI_Scatter(
    all_data, ?, MPI_INT, // send
    data, ?, MPI_INT,     // recv
    root, MPI_COMM_WORLD);
```

# Frage

Lösung:

```
int all_data[] = { 2, 3, 5, 7, 11, 13 }; // on root
int *data = malloc(2 * sizeof(int));
MPI_Scatter(
    all_data, 2, MPI_INT, // send
    data, 2, MPI_INT,     // recv
    root, MPI_COMM_WORLD);
```

# Aufgabe 4

- Erklären Sie Ihrem Sitznachbarn / Ihrer Sitznachbarin die Funktionsweise des Programms.
- Besprechen Sie gemeinsam, welche kollektive Operation als Ersatz für die Zeilen 21 – 29 benutzt werden könnte.
- Vergleichen Sie Ihren Programmcode für Teilaufgabe 3.

# Aufgabe 5

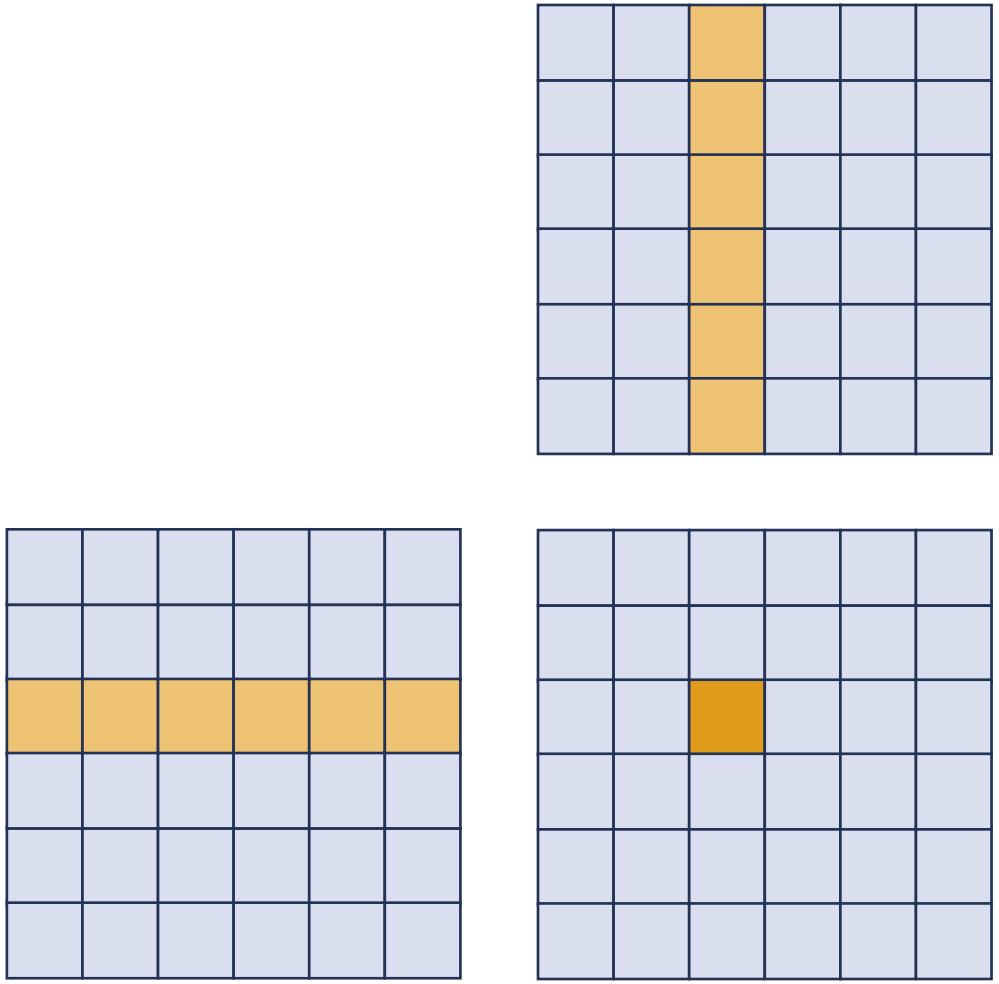
*Siehe Übungsblatt.*

# Aufgabe 6

*Siehe Übungsblatt.*

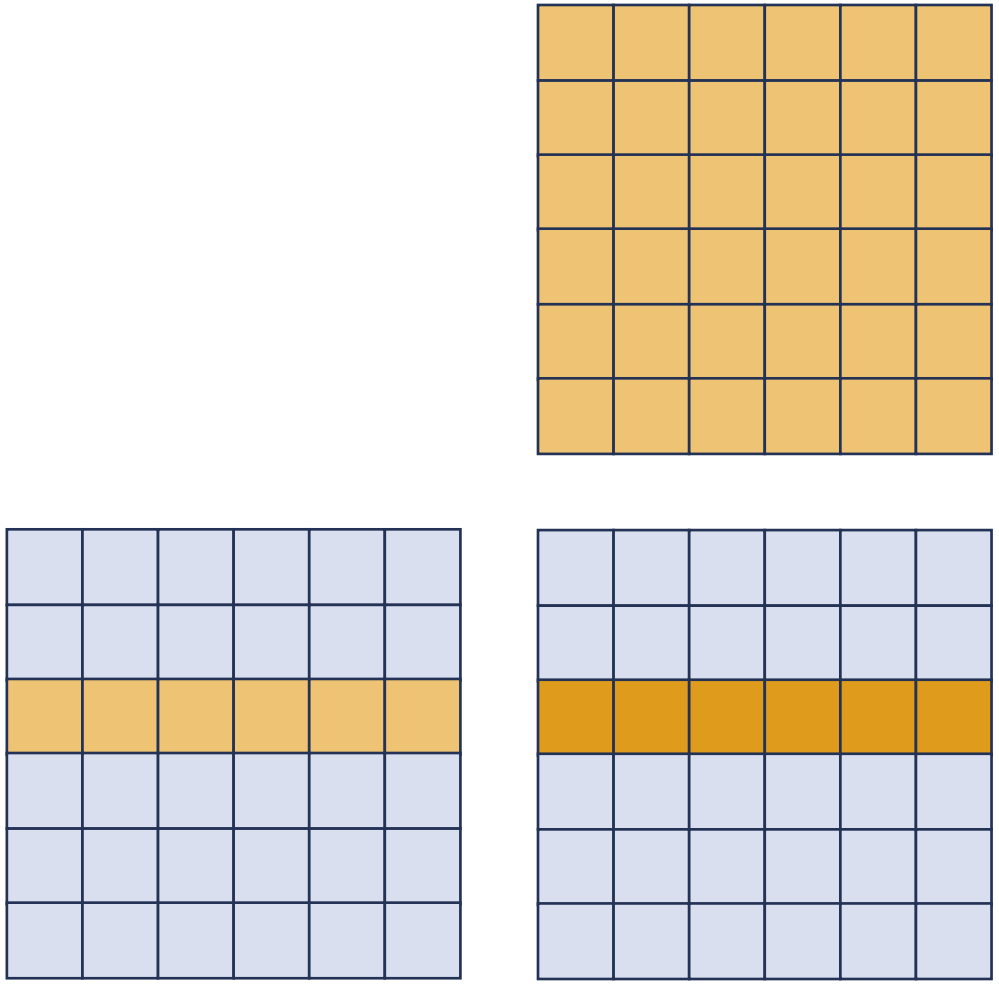
# Aufgabe 6

Matrixmultiplikation:



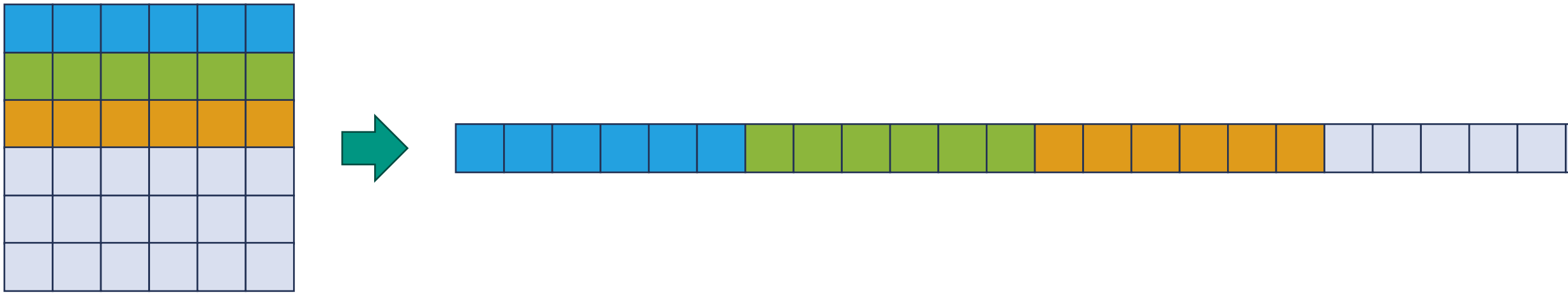
# Aufgabe 6

Matrixmultiplikation:



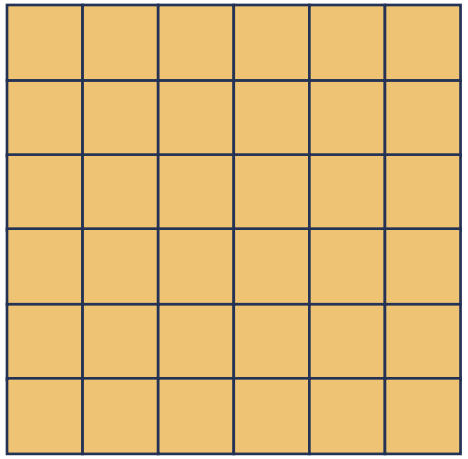
# Aufgabe 6

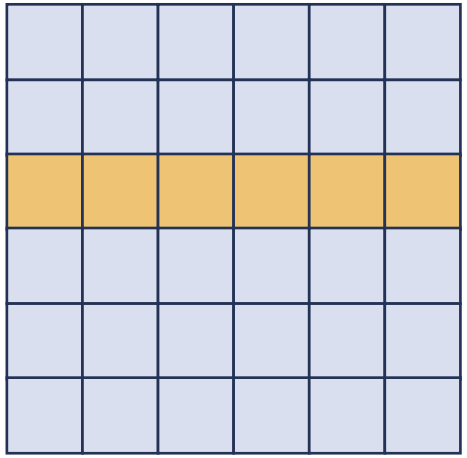
Matrizen im Speicher:

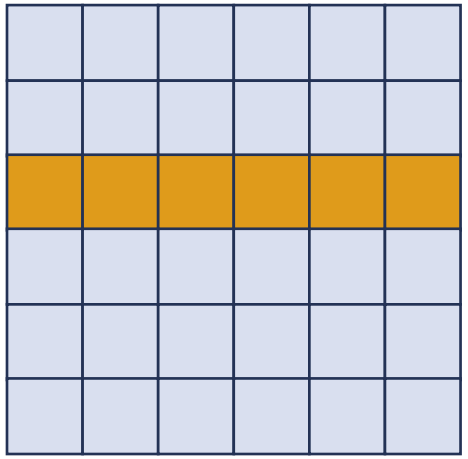


# Aufgabe 6

Welche kollektiven Operationen eignen sich hier?







# Zusammenfassung

- Task- vs. Datenparallelismus
- Shared vs. Distributed Memory

- MPI
  - Kommunikations- und Aufrufmodi
  - Direkte und kollektive Operationen

- *Selbst ausprobieren!*