Karlsruhe Institute of Technology

Programmierparadigmen
Übung 10: C/C++ und Parallelprogrammierung

Prof. Dr. Ralf Reussner / M.Sc. Larissa Schmid

25.01.2024

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

Überblick heutige Übung

- Task-/Datenparallelismus
  - Übungsblatt Aufgabe 1
  - Alte Klausuraufgabe (SoSe 2021)
- MPI
  - Übungsblatt Aufgabe 4
  - Übungsblatt Aufgabe 5

Problemaufteilung

- 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

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Verwenden von Daten- oder Taskparallelismus?**

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Verwenden von Daten- oder Taskparallelismus?**

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Verwenden von Daten- oder Taskparallelismus?**

- Faltungsmatrizen berechnen für jeden Pixel des Bildes Farbgradienten
  → dieselbe Aufgabe pro Pixel

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Verwenden von Daten- oder Taskparallelismus?**

- Faltungsmatrizen berechnen für jeden Pixel des Bildes Farbgradienten → dieselbe Aufgabe pro Pixel
- Die Faltungsmatrix kann direkt die vorliegenden Pixel zur Berechnung verwenden

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Verwenden von Daten- oder Taskparallelismus?**

- Faltungsmatrizen berechnen für jeden Pixel des Bildes Farbgradienten → dieselbe Aufgabe pro Pixel
- Die Faltungsmatrix kann direkt die vorliegenden Pixel zur Berechnung verwenden

→ Datenparallelismus

Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Wie beschleunigen durch Parallelisierung?**

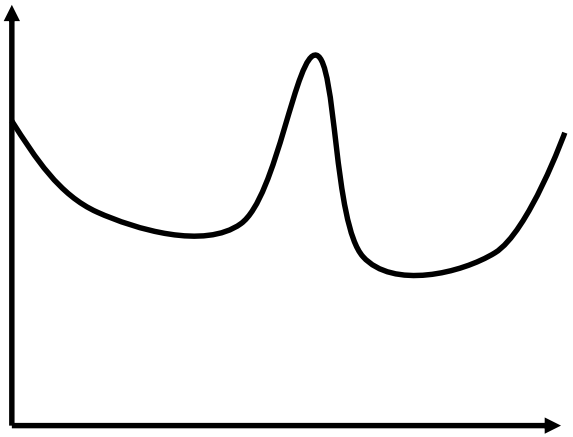
Parallelisierung von Kantendetektion
(Aufgabe 1)

- Szenario: 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.
- **Wie beschleunigen durch Parallelisierung?**

- Bild wird in mehrere Blöcke aufgeteilt
- Pro Block führt ein Thread die Berechnung des Farbgradienten für jeden Pixel durch
- Alle Farbgradienten der Blöcke werden dann zum gefilterten Bild zusammengefügt

Parallelisierung von Integral-Berechnung
(Aufgabe 1)

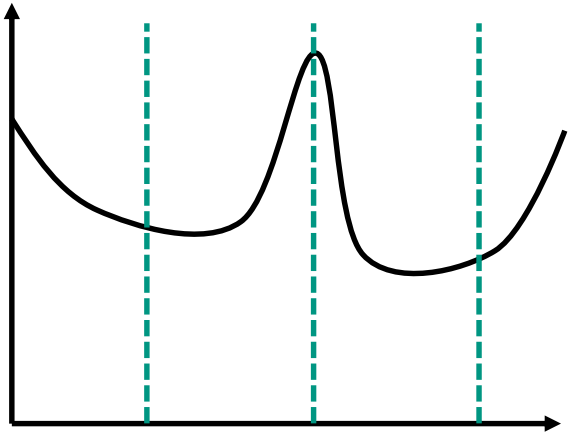
- Szenario: Es soll ein Integral numerisch berechnet werden.
- **Daten- oder Taskparallelismus?**
- **Wie beschleunigen durch Parallelisierung?**



Parallelisierung von Integral-Berechnung (Aufgabe 1)

- Szenario: Es soll ein Integral numerisch berechnet werden.
- **Daten- oder Taskparallelismus?**
- **Wie beschleunigen durch Parallelisierung?**

- Datenparallelismus
- Kurve in Teilabschnitte unterteilen
- Pro Teilabschnitt führt ein Thread die Berechnung des Integrals durch
- Alle Teilergebnisse werden dann zusammengefügt



Parallelisierung von Webserver
(Aufgabe 1)

- Szenario: Es soll ein Webserver programmiert werden, der mehrere Anfragen bearbeiten kann.
- **Daten- oder Taskparallelismus?**
- **Wie beschleunigen durch Parallelisierung?**

Parallelisierung von Webserver
(Aufgabe 1)

- Szenario: Es soll ein Webserver programmiert werden, der *mehrere Anfragen* bearbeiten kann.
- **Daten- oder Taskparallelismus?**
- **Wie beschleunigen durch Parallelisierung?**

Parallelisierung von Webserver
(Aufgabe 1)

- Szenario: Es soll ein Webserver programmiert werden, der *mehrere Anfragen* bearbeiten kann.
- **Daten- oder Taskparallelismus?**
- **Wie beschleunigen durch Parallelisierung?**

- Taskparallelismus – Anfragen können unterschiedliche Aufgaben beinhalten
- Jede Anfrage kann einem neuen Thread zugewiesen werden

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

- Szenario: Um die Helligkeit eines Graustufenbilds zu verändern, muss der Helligkeitswert jedes einzelnen Pixels verändert werden. Die Pixelinformationen eines Bilds können dabei als zweidimensionales Array dargestellt werden. In diesem soll auf jedes Element dieselbe Funktion angewendet werden. Nach Fertigstellen der Berechnung soll das veränderte Bild gespeichert werden. Eine mögliche sequentielle Implementierung für ein Bild der Größe *NxN*, gespeichert in Array *A*, könnte wie folgt aussehen. Gehen Sie davon aus, dass die Methode nur mit validen Eingaben aufgerufen wird.

```
1  Image_changeBrightness(int A[][], int amount) {
2      do i = 1..N
3          do j = 1..N
4              A[i, j] = ChangeBrightness(A[i, j], amount)
5          od
6      od
7      save(A)
8  }
9
10 ChangeBrightness(int P, int amount) {
11     return P + amount
12 }
```

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

- Szenario: Um die Helligkeit eines Graustufenbilds zu verändern, muss der Helligkeitswert jedes einzelnen Pixels verändert werden. Die Pixelinformationen eines Bilds können dabei als zweidimensionales Array dargestellt werden. In diesem soll auf jedes Element dieselbe Funktion angewendet werden. Nach Fertigstellen der Berechnung soll das veränderte Bild gespeichert werden. Eine mögliche sequentielle Implementierung für ein Bild der Größe *NxN*, gespeichert in Array *A*, könnte wie folgt aussehen. Gehen Sie davon aus, dass die Methode nur mit validen Eingaben aufgerufen wird.
- **Daten- oder Taskparallelismus?**

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

- Szenario: Um die Helligkeit eines Graustufenbilds zu verändern, muss der Helligkeitswert jedes einzelnen Pixels verändert werden. Die Pixelinformationen eines Bilds können dabei als zweidimensionales Array dargestellt werden. In diesem soll auf jedes Element dieselbe Funktion angewendet werden. Nach Fertigstellen der Berechnung soll das veränderte Bild gespeichert werden. Eine mögliche sequentielle Implementierung für ein Bild der Größe *NxN*, gespeichert in Array *A*, könnte wie folgt aussehen. Gehen Sie davon aus, dass die Methode nur mit validen Eingaben aufgerufen wird.
- **Daten- oder Taskparallelismus?**

- Daten-Parallelismus
- Es gibt nur eine Aufgabe, Anwenden der Helligkeitsfunktion *ChangeBrightness,* die gleichzeitig auf den Elementen des Arrays durchgeführt werden soll.

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

```
1  Image_changeBrightness(int A[][], int amount) {
2      do i = 1..N
3          do j = 1..N
4              A[i, j] = ChangeBrightness(A[i, j], amount)
5          od
6      od
7      save(A)
8  }
9
10 ChangeBrightness(int P, int amount) {
11     return P + amount
12 }
```

- **Wie parallelisieren? Was sind Synchronisationsbedarfe?**
  Die sequentielle Funktion *save(A)* soll nicht modifiziert werden.

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

```
1  Image_changeBrightness(int A[][], int amount) {
2      do i = 1..N
3          do j = 1..N
4              A[i, j] = ChangeBrightness(A[i, j], amount)
5          od
6      od
7      save(A)
8  }
9
10 ChangeBrightness(int P, int amount) {
11     return P + amount
12 }
```

- **Wie parallelisieren? Was sind Synchronisationsbedarfe?**
  Die sequentielle Funktion *save(A)* soll nicht modifiziert werden.
- Anwenden der *changeBrightness*-Funktion auf die einzelnen Pixel parallelisieren
- Für Berechnung keine Synchronisation notwendig
- Vor dem Aufruf von *save(A)* muss allerdings sichergestellt sein, dass alle Pixel bearbeitet wurden.

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

```
1  Image_changeBrightness(int A[][], int amount) {
2      do i = 1..N
3          do j = 1..N
4              A[i, j] = ChangeBrightness(A[i, j], amount)
5          od
6      od
7      save(A)
8  }
9
10 ChangeBrightness(int P, int amount) {
11     return P + amount
12 }
```

- **Liefert Image_changeBrightness dasselbe Ergebnis bei paralleler Ausführung auf mehreren Bildern?**

Parallelisierung von Helligkeitsberechnung
(Klausuraufgabe SoSe 2021)

```
1  Image_changeBrightness(int A[][], int amount) {
2      do i = 1..N
3          do j = 1..N
4              A[i, j] = ChangeBrightness(A[i, j], amount)
5          od
6      od
7      save(A)
8  }
9
10 ChangeBrightness(int P, int amount) {
11     return P + amount
12 }
```

- **Liefert Image_changeBrightness dasselbe Ergebnis bei paralleler Ausführung auf mehreren Bildern?**
- Ja, da es keine Abhängigkeiten/konkurrierenden Zugriffe gibt.

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

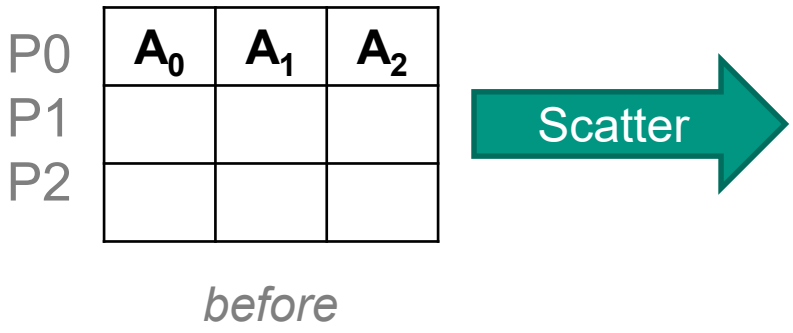
```c
void operate(int elementCount, int elements[elementCount]) {
    int rank;
    int processCount;
    int rootRank = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processCount);

    int elementsPerProcess = elementCount / processCount;
    int local[elementsPerProcess];
    MPI_Scatter(elements, elementsPerProcess, MPI_INT, local,
        elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);

    int offset = 0;
    if (rank == rootRank) {
        for (int i = 0; i < elementCount; i++) {
            if (elements[i] < offset) offset = elements[i];
        }
    }

    if (rank == rootRank) {
        for (int process = 0; process < processCount; process++) {
            if (process != rootRank) {
                MPI_Send(&offset, 1, MPI_INT, process, 0,
                    MPI_COMM_WORLD);
            }
        }
    } else {
        MPI_Recv(&offset, 1, MPI_INT, rootRank, 0, MPI_COMM_WORLD, NULL);
    }

    for (int i = 0; i < elementsPerProcess; i++) {
        local[i] = local[i] - offset;
    }

    MPI_Gather(local, elementsPerProcess, MPI_INT, elements,
        elementsPerProcess, MPI_INT, rootRank, MPI_COMM_WORLD);

    if (rank == rootRank) {
        for (int i = 0; i < elementCount; i++) {
            printf("%i ", elements[i]);
        }
    }
```

- Welche Funktion erfüllt das Programm?
- Was ist die Ausgabe für die Eingabe (4, [-1, 2, 9, 5])?

MPI_Scatter

```
int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

- All receivers get equal-sized but *content-different* data
  - ➔ scatter and broadcast are different
- `sendcount` and `recvcount` are the numbers of elements sent to and received by one process and are usually equal

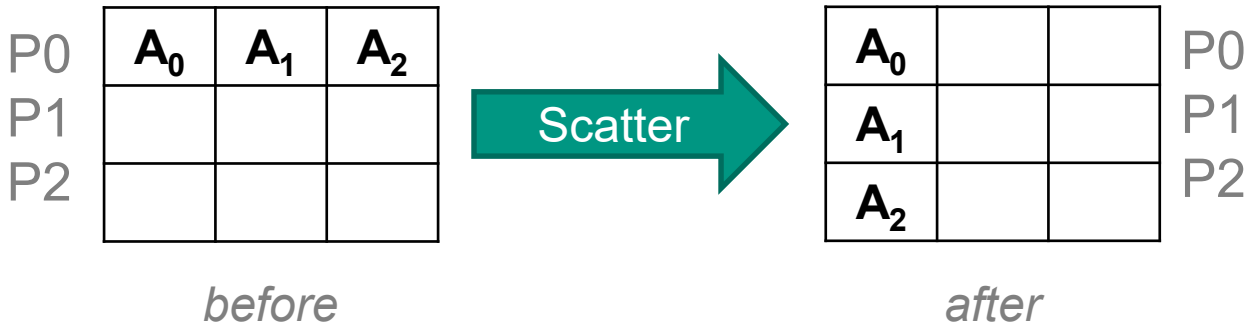


more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html#Node72

MPI_Scatter

```
int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

- All receivers get equal-sized but *content-different* data
  - ➔ scatter and broadcast are different
- `sendcount` and `recvcount` are the numbers of elements sent to and received by one process and are usually equal

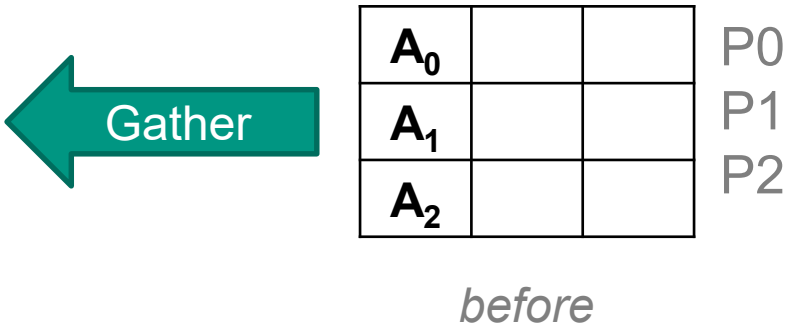


more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html#Node72

MPI_Gather

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

- root's buffer contains collected data *sorted by rank*, including root's own buffer contents
- Receive buffer is ignored by all non-root processes
- recvcount: number of items received from ***each*** process
- Recall that MPI_Scatter is the inverse of MPI_Gather



- MPI_Gatherv is the vector variant again

more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70

MPI_Gather

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

- root's buffer contains collected data *sorted by rank*, including root's own buffer contents
- Receive buffer is ignored by all non-root processes
- recvcount: number of items received from ***each*** process
- Recall that MPI_Scatter is the inverse of MPI_Gather

- MPI_Gatherv is the vector variant again

more examples: http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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);
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

Eingabe (4, [-1, 2, 9, 5])

```
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);
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

Eingabe (4, [-1, 2, 9, 5])

```
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);
```

Annahme: 2

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

Eingabe (4, [-1, 2, 9, 5])

```
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, &proce
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);
```

Annahme: 2

elementsPerProcess = 4 / 2 = 2

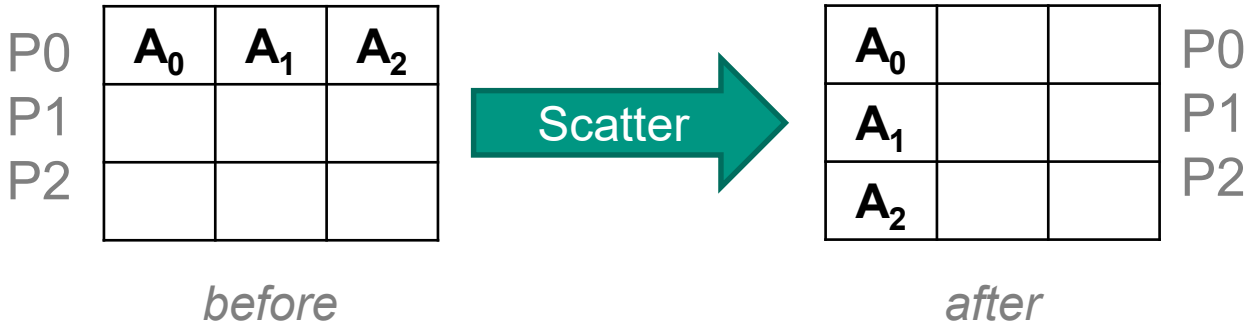
MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

Eingabe (4, [-1, 2, 9, 5])

```
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, &proce
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);
```

Annahme: 2

elementsPerProcess = 4 / 2 = 2



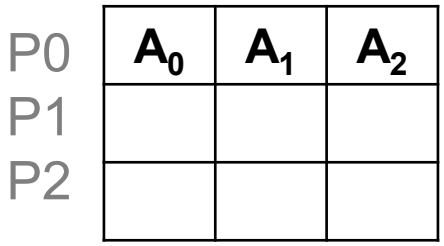
MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

Eingabe (4, [-1, 2, 9, 5])

```
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, &proce
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);
```

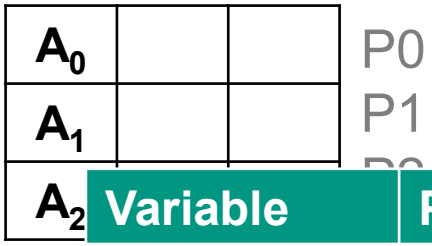
Annahme: 2

elementsPerProcess = 4 / 2 = 2



before

Scatter



Variable	Prozess 0	Prozess 1
elements	[-1, 2, 9, 5]	[-1, 2, 9, 5]
local	[-1, 2]	[9, 5]

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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     }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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,
8
9      int elementsPerProcess = eler
10     int local[elementsPerProcess
11     MPI_Scatter(elements, elemen
12         elementsPerProcess, MPI_
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     }
```

Variable	Prozess 0	Prozess 1
elements	[-1, 2, 9, 5]	[-1, 2, 9, 5]
local	[-1, 2]	[9, 5]
offset	-1	0

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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      }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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 < elementsPerProces
32          local[i] = local[i] - offset;
33      }
34
35      MPI_Gather(local, elementsPerProcess,
36          elementsPerProcess, MPI_INT, root
37
38      if (rank == rootRank) {
39          for (int i = 0; i < elementCount; i++) {
40              printf("%i␣", elements[i]);
41          }
42      }
```

Variable	Prozess 0	Prozess 1
elements	[-1, 2, 9, 5]	[-1, 2, 9, 5]
local	[-1, 2]	[9, 5]
offset	-1	-1

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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      }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
21      if (rank == rootRank) {
22          for (int process = 0; process < processCount; process++) {
23              if (process != rootRank) {
24                  MPI_Send(&offset, 1, MPI_
                        MPI_COMM_WORLD);
25              }
26          }
27      } else {
28          MPI_Recv(&offset, 1, MPI_INT, roc
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      }
```

Variable	Prozess 0	Prozess 1
elements	[-1, 2, 9, 5]	[-1, 2, 9, 5]
local	[0, 3]	[10, 6]
offset	-1	-1

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
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      }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
21      if (rank == rootRank) {
22          for (int process = 0; process < processCount; process++) {
23              if (process != rootRank) {
24                  MPI_Send(&offset, 1, MPI_
                        MPI_COMM_WORLD);
25              }
26          }
27      } else {
28          MPI_Recv(&offset, 1, MPI_INT, ro
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      }
```

Variable	Prozess 0	Prozess 1
elements	[0, 3, 10, 6]	[-1, 2, 9, 5]
local	[0, 3]	[10, 6]
offset	-1	-1

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

```
21      if (rank == rootRank) {
22          for (int process = 0; process < processCount; process++) {
23              if (process != rootRank) {
24                  MPI_Send(&offset, 1, MPI
                        MPI_COMM_WORLD);
25              }
26          }
27      } else {
28          MPI_Recv(&offset, 1, MPI_INT, rc
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      }
```

Variable	Prozess 0	Prozess 1
elements	**[0, 3, 10, 6]**	[-1, 2, 9, 5]
local	[0, 3]	[10, 6]
offset	-1	-1

- Welche Funktion erfüllt das Programm?
- Erhöhen aller Werte von elements um den kleinsten Wert des Arrays, wenn dieser kleiner als null ist.

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

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

```
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      }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

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

```
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      }
```

- *MPI_Bcast(&offset, 1, MPI_INT, rootRank, MPI_COMM_WORLD);*

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

- Die Zeilen 15 bis 19 werden rein sequentiell ausgeführt. Wie können sie möglichst effizient mittels MPI parallelisiert werden?

```
15      if (rank == rootRank) {
16          for (int i = 0; i < elementCount; i++) {
17              if (elements[i] < offset) offset = elements[i];
18          }
19      }
```

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

- Die Zeilen 15 bis 19 werden rein sequentiell ausgeführt. Wie können sie möglichst effizient mittels MPI parallelisiert werden?

```
15      if (rank == rootRank) {
16          for (int i = 0; i < elementCount; i++) {
17              if (elements[i] < offset) offset = elements[i];
18          }
19      }
```

- Lokal in jedem Prozess für sein Teilarray *local* Minimalwert berechnen

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

- Die Zeilen 15 bis 19 werden rein sequentiell ausgeführt. Wie können sie möglichst effizient mittels MPI parallelisiert werden?

```
15      if (rank == rootRank) {
16          for (int i = 0; i < elementCount; i++) {
17              if (elements[i] < offset) offset = elements[i];
18          }
19      }
```

- Lokal in jedem Prozess für sein Teilarray *local* Minimalwert berechnen
- Globales Minimum per *MPI_Reduce* ermitteln

MPI: Von Send/Receive zu kollektiven Operationen (Übungsblatt Aufgabe 4)

- Die Zeilen 15 bis 19 werden rein sequentiell ausgeführt. Wie können sie möglichst effizient mittels MPI parallelisiert werden?

```
15      if (rank == rootRank) {
16          for (int i = 0; i < elementCount; i++) {
17              if (elements[i] < offset) offset = elements[i];
18          }
19      }
```

- Lokal in jedem Prozess für sein Teilarray *local* Minimalwert berechnen
- Globales Minimum per *MPI_Reduce* ermitteln

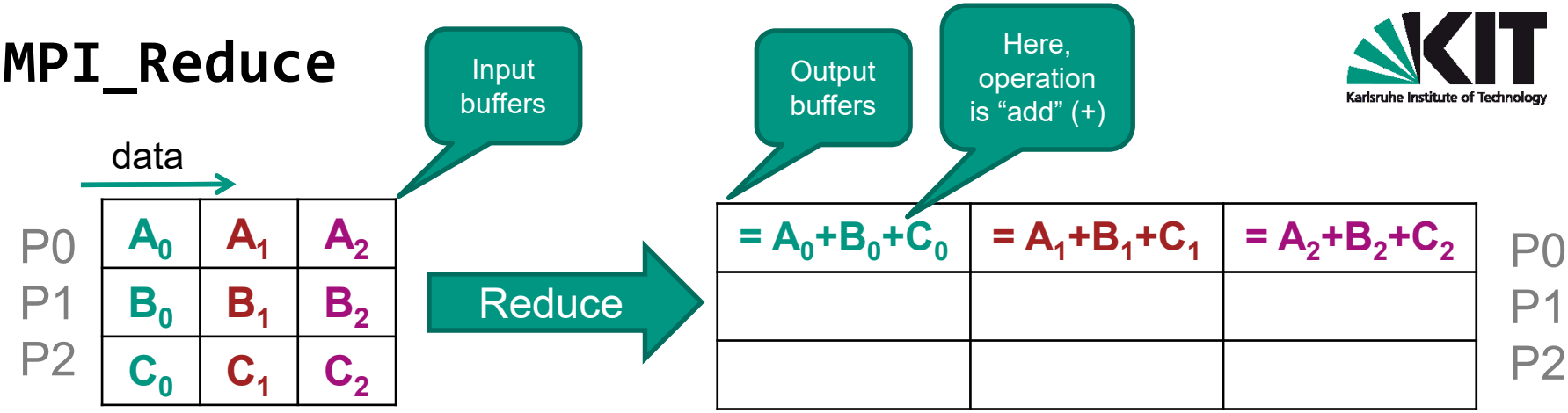
```
int localOffset = 0;
for (int i = 0; i < elementsPerProcess; i++) {
    if (local[i] < localOffset) localOffset = local[i];
}
MPI_Reduce(&localOffset, &offset, 1, MPI_INT, MPI_MIN,
    rootRank, MPI_COMM_WORLD);
```

MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

- 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);
```

MPI_Reduce

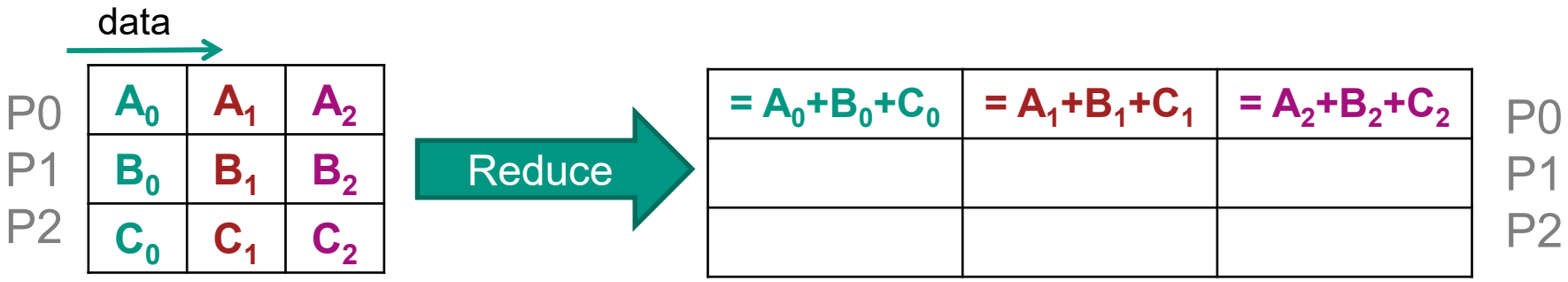



```
int MPI_Reduce( void* sendbuf, void* recvbuf, int count,
                MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
```

- Applies an operation to the data in `sendbuf` and stores the result in `recvbuf` of the root process
- `count`: number of columns in the output buffer
- `op`: can be …
  - logical / bitwise “and” / “or”: `MPI_LAND` / `MPI_BAND` / `MPI_LOR` / `MPI_BOR` / ...
  - `MPI_MAX` / `MPI_MIN` / `MPI_SUM` / `MPI_PROD` / ...
  - `MPI_MINLOC` / `MPI_MAXLOC` find local minimum / maximum and return the value of the "causing" rank
  - own operations can also be defined

MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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);
```



MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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);
```

Puffer	Rank	Inhalt
sendbuffer	0	0, 2, 4, 6
	1	1, 3, 5, 7
	2	2, 4, 6, 8
	3	3, 5, 7, 9
recvbuffer	0	6, 14, 22, 30

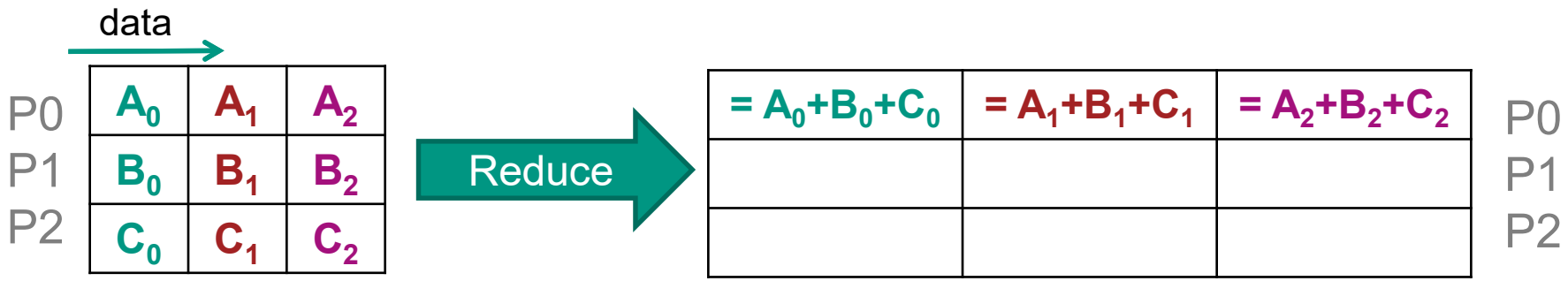
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

- 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, MPI_Comm_rank*

- Verwenden Sie die folgende Signatur:
  ***void** my_int_sum_reduce(**int** *sendbuf, **int** *recvbuf, **int** count, **int** root, MPI_Comm comm)*

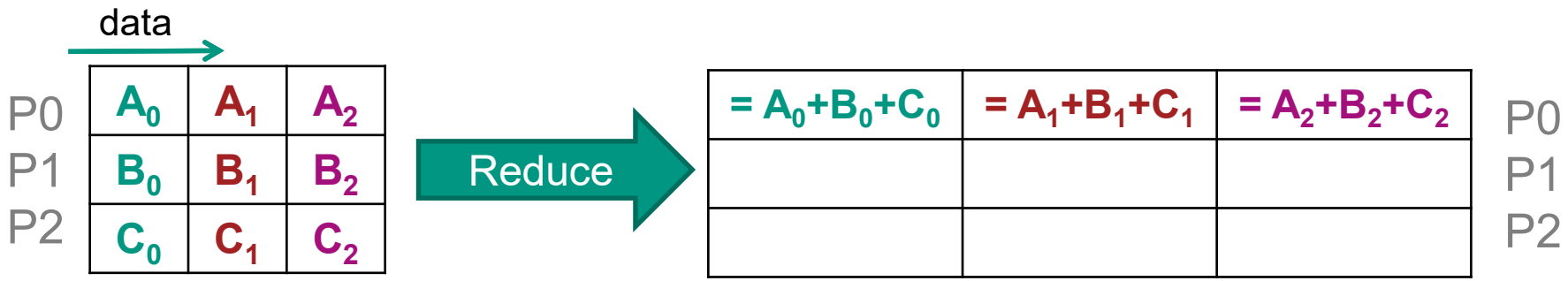
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

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



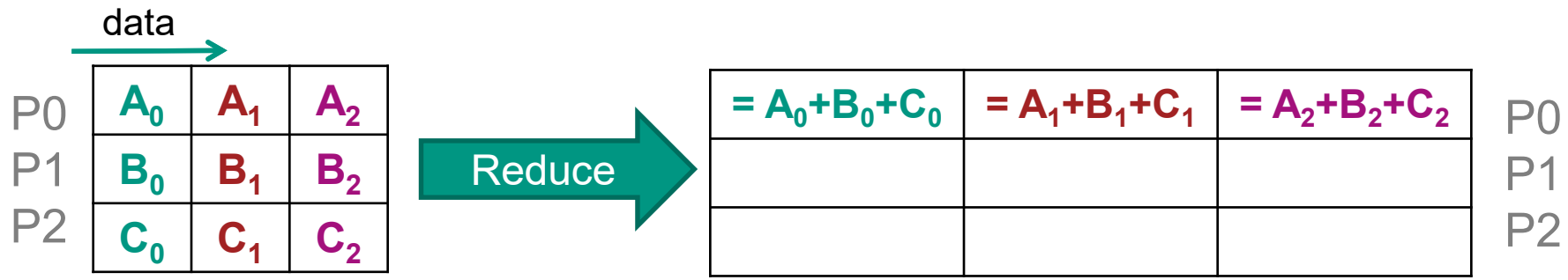
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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
```



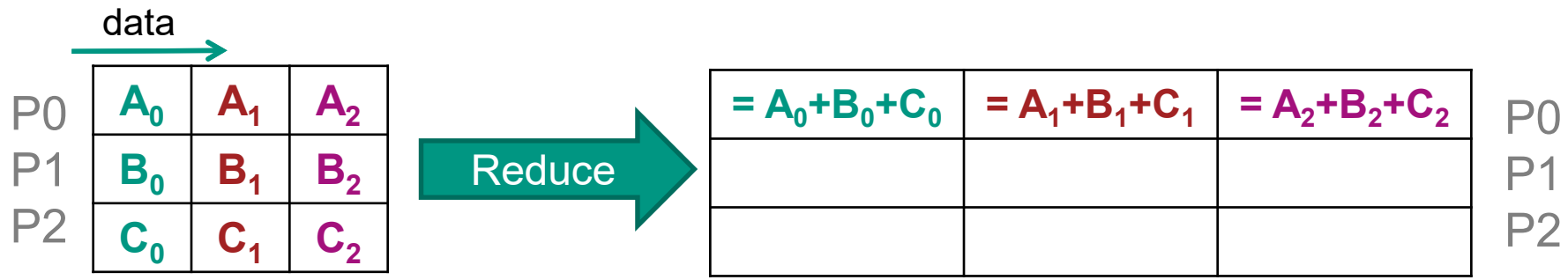
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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
```



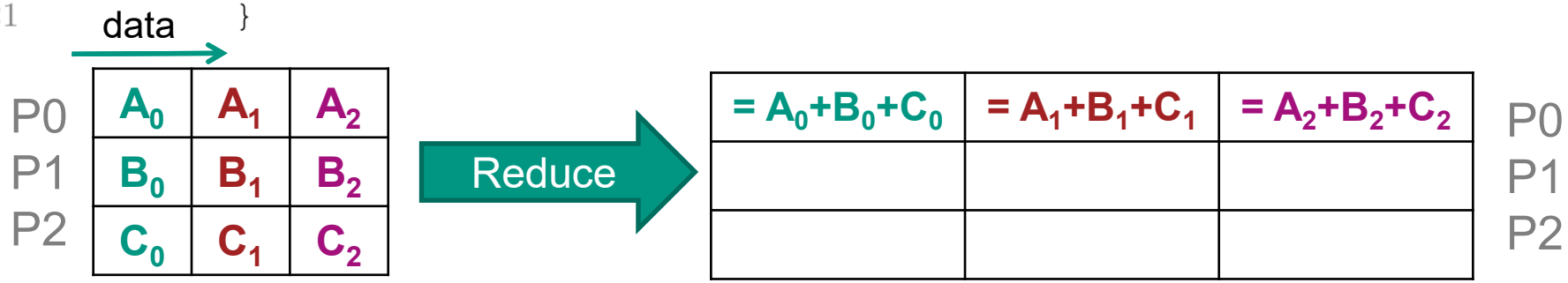
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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;
```



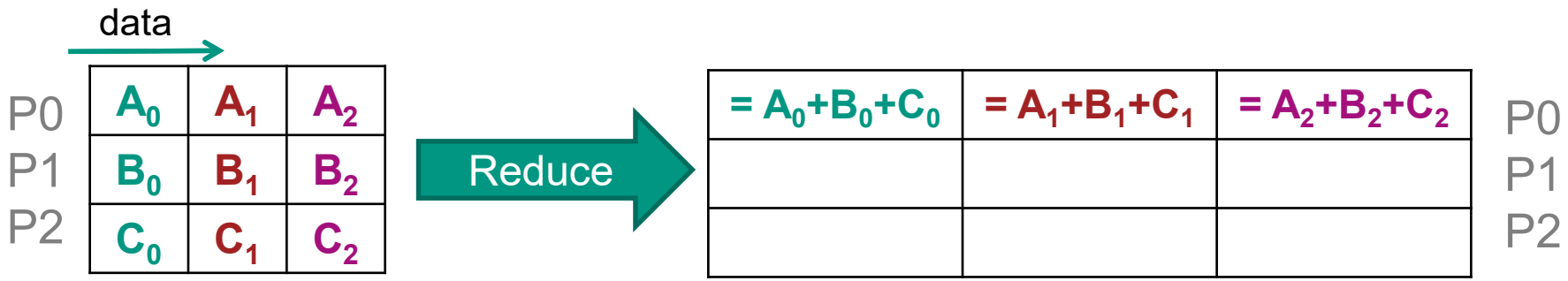
MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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          }
```



MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
22        } else {
23            /* Send our values to root. */
24            MPI_Send(sendbuf, count, MPI_INT, root, 0, comm);
25        }
26    }
```



MPI: Reduce und der Weg zurück zu Send/Receive (Übungsblatt Aufgabe 5)

```
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 }
```