

# Tutorium 10: Parallelprogrammierung mit MPI

---

Paul Brinkmeier

17. Januar 2023

Tutorium Programmierparadigmen am KIT

# Heutiges Programm

---

ProPa-Stoff zu Parallelprogrammierung:

- Grundlegende Begriffe
- Message Passing, wurde in OS *kurz* behandelt („message queues“)
- Shared Memory + Synchronisierung, wie in SWT1, OS, etc.
  - Mit ein paar Java-Details

# Begriffe

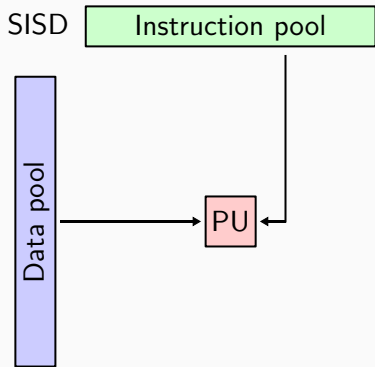
---

- SISD: Single Instruction, Single Data  
Ein Datum wird von einer Ausführungsarbeit bearbeitet
- SIMD: Single Instruction, Multiple Data  
Eine Ausführungseinheit bearbeitet mehrere Daten gleichzeitig
- MIMD: Multiple Instruction, Multiple Data  
≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig
- MISD: Multiple Instruction, Single Data  
≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig an einem Datum

- SISD: Single Instruction, Single Data  
Ein Datum wird von einer Ausführungsarbeit bearbeitet
- SIMD: Single Instruction, Multiple Data  
Eine Ausführungseinheit bearbeitet mehrere Daten gleichzeitig
- MIMD: Multiple Instruction, Multiple Data  
≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig
- MISD: Multiple Instruction, Single Data  
≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig an einem Datum

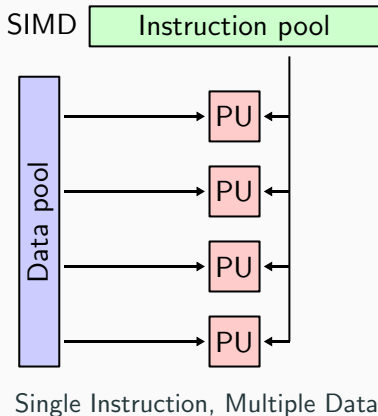
Beispiele?

# SISD



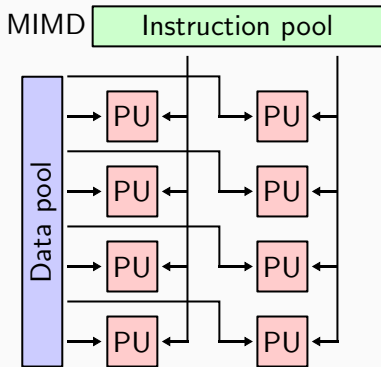
Single Instruction, Single Data

# SIMD



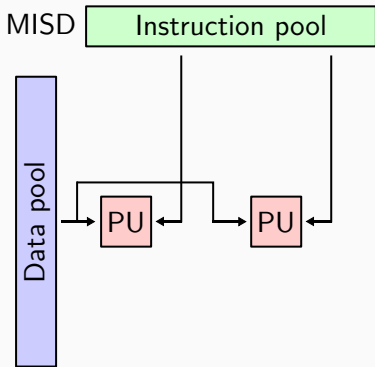


# MIMD



Multiple Instruction, Multiple Data

# MISD



Multiple Instruction, Single Data

Parallele Probleme sind üblicherweise entweder

- „datenparallel“: Problem kann auf identische Ausführungseinheiten verteilt werden  
Beispiel: `map primeFactors [1432793, 651433, ...]`
- „taskparallel“: Problembestandteile sind nicht homogen  
Beispiel: Videospiel mit Render-, Netzwerk- und Logikprozessen

Datenparallele Probleme sind i.d.R. einfacher zu behandeln (auch: „embarrassingly parallel“). Bei manchen Problemen verschwimmt die Grenze auch (bspw. Webserver).

# MPI-Basics

---

MPI („Message Passing Interface“) ist ein Standard für Parallelprogrammierung. Es existieren verschiedene Implementierungen für verschiedene Sprachen. Die VL verwendet [Open MPI](#), eine Open-Source-Implementierung.

- MPI-„Prozesse“ beziehen sich i.d.R. auf Prozessorkerne
- Message Passing statt Shared Memory:
  - Daten werden explizit über `Send` und `Recv` geteilt
- MPI-Prozesse werden in sog. *Communicators* eingeteilt. Wir verwenden immer den Communicator, der alle Prozesse enthält (`MPI_COMM_WORLD`)

MPI-Beispiele gehen von Linux-Systemen aus, verwendet unter Windows bitte WSL.

- `apt install openmpi-bin` (Ubuntu)
- `pacman -S openmpi` (Arch Linux)
- `dnf install openmpi` (Fedora)
- `brew install open-mpi` (macOS)

Verwendet `mpicc --version` zum Testen der Installation.

# MPI: Grundgerüst

```
#include <mpi.h>

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

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

    ...

    MPI_Finalize();
}
```

Grundlegende Begriffe:

- *Communicator*: Gruppe von Prozessen
- `MPI_COMM_WORLD`: Communicator, der alle Prozesse enthält
- `size`: Gesamtzahl der Prozesse
- `rank`: Laufende Nummer eines Prozesses  $\in [0, \text{size})$
- `root`: Ausgangspunkt einer kollektiven Operation



# Bauen und Ausführen von MPI-Programmen

MPI-Programme werden mit `mpicc` (Wrapper um `gcc` oder `clang`) kompiliert:

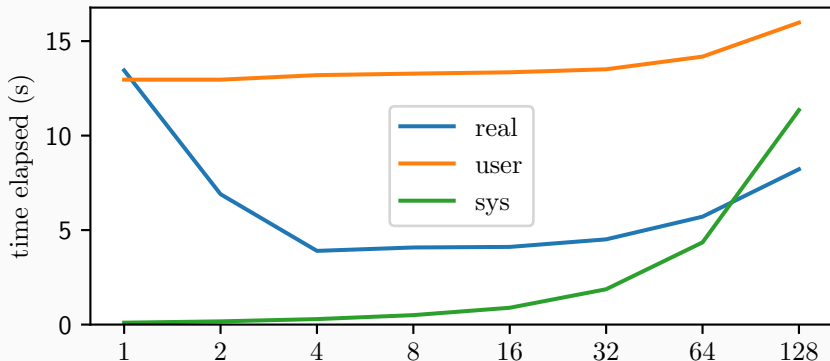
```
cd demos/mpi/hello  
mpicc -o hello hello.c # oder make
```

Um ein Programm auszuführen, wird `mpirun` verwendet:

```
mpirun --host localhost:N ./hello
```

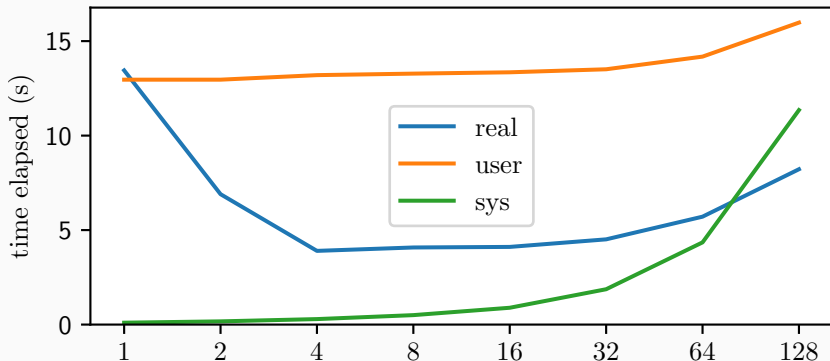
- `N` ist die Zahl der Prozesse, die ausgeführt werden sollen
- Betrachtet die Demos `mpi/hello` und `mpi/sendrecv`.

## Beispielausführung von `mpi/hello`



- *real*: Tatsächlich vergangene Zeit
- *user/sys*: Auf Prozessoren vergangene Zeit (Im User- bzw. Kernelmode)

## Beispielausführung von mpi/hello



- *real*: Tatsächlich vergangene Zeit
- *user/sys*: Auf Prozessoren vergangene Zeit (Im User- bzw. Kernelmode)
- Prozessor: 4 × Intel Core i5-7600K @ 3,8GHz

Mit den Message-Passing-Primitiven Send und Recv werden Daten zwischen Prozessen ausgetauscht.

Die Aufrufe sind unabhängig vom Medium (IPC, Sockets, ...).



- `int MPI_Send(buf, count, datatype, dest, tag, comm)`
- `int MPI_Recv(buf, count, datatype, source, tag, comm, status)`

# Kollektive Operationen

---

Bcast verteilt ein Datum auf alle Prozesse.



- `int MPI_Bcast(buf, count, datatype, root, comm)`
- Daten befinden sich ursprünglich auf root
  - $\leadsto$  Fallunterscheidung in Bcast:
  - `if rank == root then forall others: send() else recv()`

# Bcast

Bcast verteilt ein Datum auf alle Prozesse.



- `int MPI_Bcast(buf, count, datatype, root, comm)`
- Daten befinden sich ursprünglich auf root
  - $\leadsto$  Fallunterscheidung in Bcast:
  - `if rank == root then forall others: send() else recv()`

Implementiert `custom_Bcast` in `demos/mpi/custom_broadcast!`

Scatter verteilt eine Liste von Daten auf mehrere Prozesse.



- `int MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`
- `sendcount, recvcount`: Zahl der Elemente, die an einen Prozess verteilt werden
- l.d.R.: `sendcount == recvcount`



Gather sammelt Daten von allen Prozessen in einer Liste.



- `int MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`
- `sendcount, recvcount`: Zahl der Elemente, die an einen Prozess verteilt werden
- I.d.R.: `sendcount == recvcount`

# Scatter und Gather

Scatter und Gather sind mehr oder weniger „invers“:

```
int nums[4];
int local;
if (rank == 0) { nums = {0, 1, 2, 3}; }

MPI_Scatter(nums, 1, MPI_INT, &local, 1, MPI_INT,
            0, MPI_COMM_WORLD);
// in P_i gilt: local = i
MPI_Gather(&local, 1, MPI_INT, nums, 1, MPI_INT,
            0, MPI_COMM_WORLD);
```

Häufiges Muster: Scatter, Daten bearbeiten, Gather, Ergebnisse zusammenführen

# Aufgabe zu Scatter und Gather

Implementiert folgendes Programm mit MPI:

- $N$ : Prozessoranzahl (`MPI_Comm_size`),  $x = 1000$
- $P_0$  legt long-Liste mit Elementen  $[1, 2, \dots, N \cdot x]$  an
- $P_i$  summiert einen  $x$ -Ausschnitt der Liste mit  $i \in [0; N)$
- $P_0$  summiert die einzelnen Summen

Verwendet dafür:

- `MPI_Comm_size`, `MPI_Comm_rank`
- `MPI_Scatter`
- `MPI_Gather`

Dokumentation für MPI-Funktionen bekommt ihr mit `man <f>`

Allgather ist die „Verkettung“ von Gather und Bcast.



- `int MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`
- Im Gegensatz zu Gather gibt es keinen Parameter root

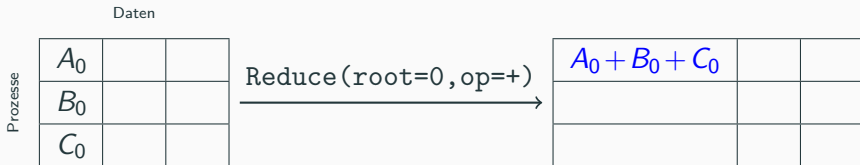
Alltoall stückelt Daten von jedem Prozess und verteilt sie.



- `int MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype, comm)`
- Es führt sozusagen jeder Prozess einmal Scatter aus

# Reduce

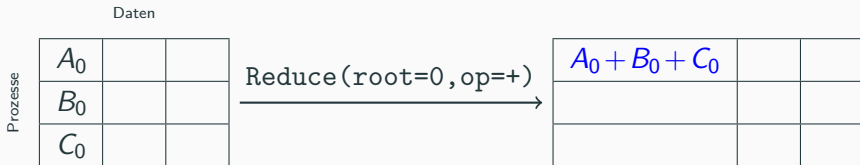
Reduce wendet eine assoziative Operation auf verteilte Daten an.



- `int MPI_Reduce(sendbuf, recvbuf, count, type, op, root, comm)`
  - Beispiele für op: `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`, etc.
- *Ungefähr* dasselbe wie ein Fold!

# Reduce

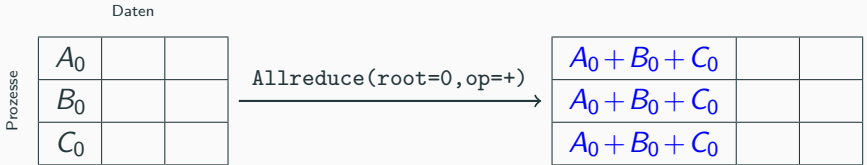
Reduce wendet eine assoziative Operation auf verteilte Daten an.



- `int MPI_Reduce(sendbuf, recvbuf, count, type, op, root, comm)`
  - Beispiele für `op`: `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`, etc.
- *Ungefähr* dasselbe wie ein Fold!
- Ersetzt den letzten Teil der Summenaufgabe durch einen Aufruf zu Reduce!

# Allreduce

Allreduce ist die Verkettung von Reduce und Bcast.



- `int MPI_Allreduce(sendbuf, recvbuf, count, type, comm, op, comm)`
- Wie bei Allgather/Alltoall: Kein root-Parameter
- Reduce und Allreduce funktionieren außerdem auch „mehrspalzig“, d.h. auch  $A_1 + B_1 + C_1$  etc.



**Ende**

---

- Im [CAS](#) könnt ihr euch bis zum 24.03. für die PP-Klausur anmelden
  - Termin: 31.03.2023
- Bis zum 15.02. könnt ihr euch [Rückmelden](#)
- Noch eine gute Woche :)