Message Passing mit modernem C++

Dr. rer. nat. Heiko Bauke

para//el 2018

Message Passing und der MPI Standard

Message Passing Library

Zusammenfassung

Message Passing und der MPI Standard

Message Passing (Interface)

- Message Passing

Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme

Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)

Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher

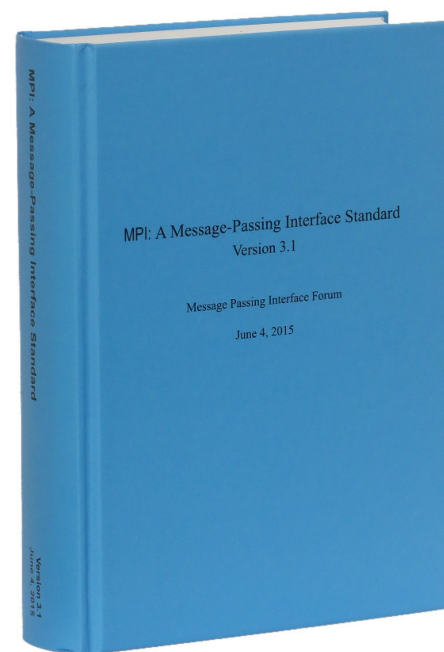
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern

Cray XC40 (Hazel Hen), High Performance Computing Center Stuttgart
Quelle: Julian Herzog via Wikimedia Commons

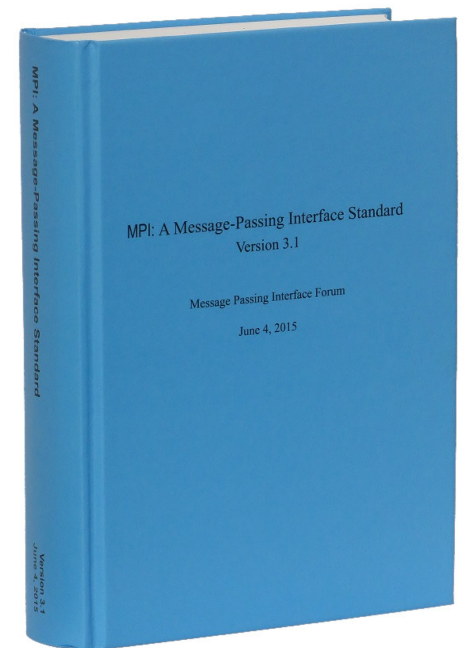
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard



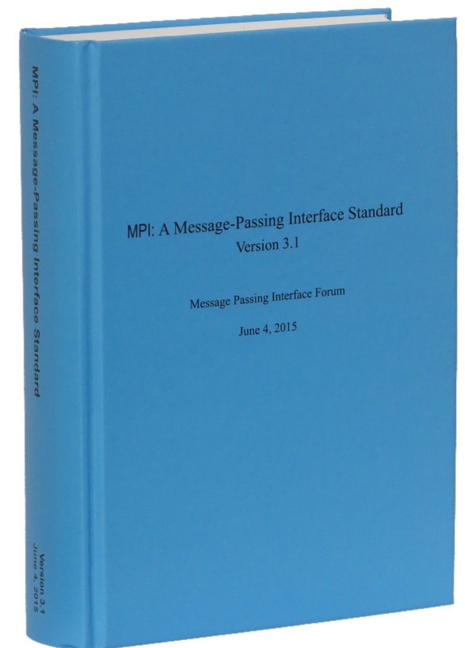
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen



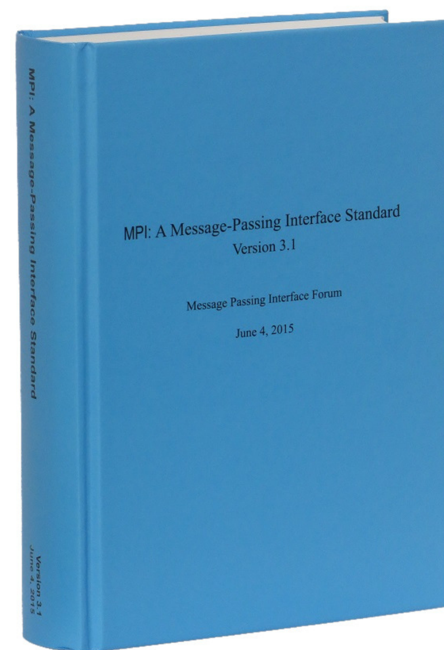
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern



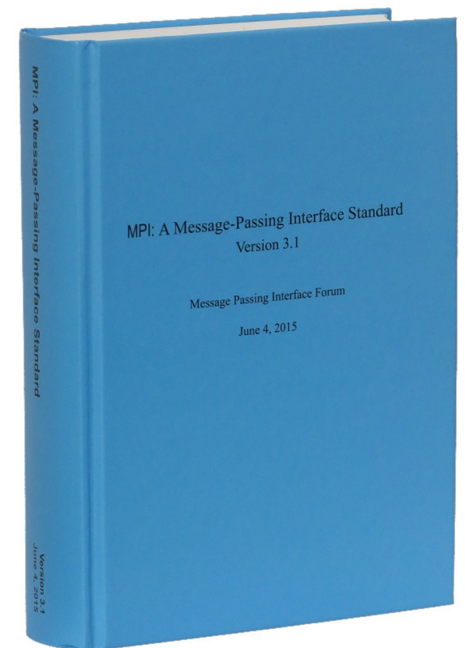
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern
  - spezifiziert durch MPI Forum (http://mpi-forum.org)



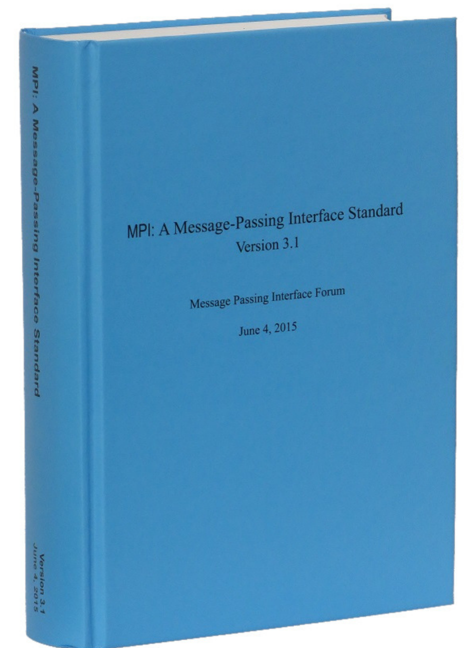
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern
  - spezifiziert durch MPI Forum (http://mpi-forum.org)
  - definiert API und deren Semantik für C und FORTRAN



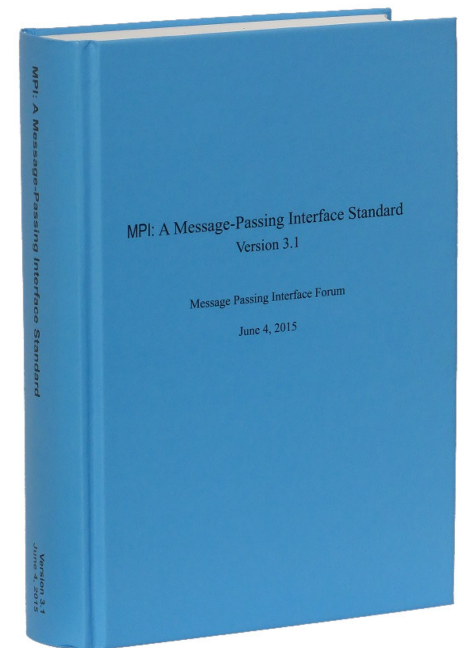
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern
  - spezifiziert durch MPI Forum (http://mpi-forum.org)
  - definiert API und deren Semantik für C und FORTRAN
  - verschiedene Implementierungen des Standards / des API (OpenMPI, MPICH, …)



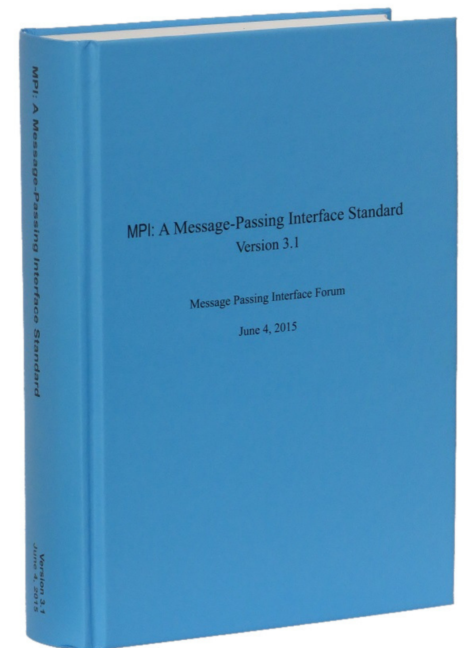
Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern
  - spezifiziert durch MPI Forum (http://mpi-forum.org)
  - definiert API und deren Semantik für C und FORTRAN
  - verschiedene Implementierungen des Standards / des API (OpenMPI, MPICH, . . .)
  - C++-API in Version 3.0 aus dem MPI Standard entfernt



Message Passing (Interface)

- Message Passing
  - Paradigma zur Programmierung und Koordinierung paralleler Programme
  - Kommunikation durch Austausch von Nachrichten (statt z.B. durch shared memory)
  - geeignet für Architekturen mit gemeinsamen oder verteiltem Speicher
  - vorherrschendes Paradigma zur Programmierung von HPC-Clustern
- Message Passing Interface (MPI) Standard
  - *Bibliotheksspezifikation* für Funktionen zum Nachrichtenaustausch in parallelen Programmen
  - *die* Grundlage zur Programmierung von HPC-Clustern
  - spezifiziert durch MPI Forum (http://mpi-forum.org)
  - definiert API und deren Semantik für C und FORTRAN
  - verschiedene Implementierungen des Standards / des API (OpenMPI, MPICH, ...)
  - C++-API in Version 3.0 aus dem MPI Standard entfernt, (mögliche) Gründe:
    - schlechtes Design des C++-API
    - mangelnde Akzeptanz bei MPI-Usern
    - fehlende Man-Power & C++-Expertise im MPI Forum



Message Passing (Interface)

Warum C++ für HPC-Anwendungen?

Warum C++ für HPC-Anwendungen?

Performance

Warum C++ für HPC-Anwendungen?

Performance

Warum C++ für HPC-Anwendungen?

I have yet to see a program that can be written better in C than in C++. [. . .] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.

Bjarne Stroustrup

Warum C++ für HPC-Anwendungen?

„Hallo Welt!“ in C

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

int main(void) {
  size_t len=0x7ffffffe;  // len = 2^31 − 2
  char *str=malloc(len+1);
  if (str==NULL)
    return EXIT_FAILURE;
  memset(str, '#', len);
  str[len]=0;
  printf("Hallo %s Welt!\n", str);
  free(str);
}
```

Warum C++ für HPC-Anwendungen?

„Hallo Welt!“ in C

```c
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

int main(void) {
  size_t len=0x7ffffffe;  // len = 2^31 − 2
  char *str=malloc(len+1);
  if (str==NULL)
    return EXIT_FAILURE;
  memset(str, '#', len);
  str[len]=0;
  printf("Hallo %s Welt!\n", str);
  free(str);
}
```

„Hallo Welt!“ in C++

```cpp
#include <string>
#include <iostream>

int main() {
  std::string str(0x7ffffffe, '#');
  std::cout << "Hallo " << str << " Welt!\n";
}
```

Warum C++ für HPC-Anwendungen?

„Hallo Welt!“ in C

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

int main(void) {
  size_t len=0x7ffffffe;  // len = 2^31 − 2
  char *str=malloc(len+1);
  if (str==NULL)
    return EXIT_FAILURE;
  memset(str, '#', len);
  str[len]=0;
  printf("Hallo %s Welt!\n", str);
  free(str);
}
```

„Hallo Welt!“ in C++

```
#include <string>
#include <iostream>

int main() {
  std::string str(0x7ffffffe, '#');
  std::cout << "Hallo " << str << " Welt!\n";
}
```

Ausgabe: „`Hallo ###. . . ### Welt!`“

Warum C++ für HPC-Anwendungen?

„Hallo Welt!“ in C

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

int main(void) {
  size_t len=0x7ffffffe;  // len = 2^31 − 2
  char *str=malloc(len+1);
  if (str==NULL)
    return EXIT_FAILURE;
  memset(str, '#', len);
  str[len]=0;
  printf("Hallo %s Welt!\n", str);
  free(str);
}
```

Ausgabe: „`Hallo ###. . . ###`“

„Hallo Welt!“ in C++

```
#include <string>
#include <iostream>

int main() {
  std::string str(0x7ffffffe, '#');
  std::cout << "Hallo " << str << " Welt!\n";
}
```

Ausgabe: „`Hallo ###. . . ### Welt!`“

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. […] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. […] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. [...] I don’t believe such a program could exist. By “better” I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. [. . .] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. [...] I don’t believe such a program could exist. By “better” I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*
  - syntaktischer Zucker (Überladen von Funktionen, Lambda-Funktionen etc.)

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. […] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*
  - syntaktischer Zucker (Überladen von Funktionen, Lambda-Funktionen etc.)
  - generische Funktionen und generische Klassen durch Templates

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. […] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*
  - syntaktischer Zucker (Überladen von Funktionen, Lambda-Funktionen etc.)
  - generische Funktionen und generische Klassen durch Templates
  - weniger *boilerplate code*

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. [...] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*
  - syntaktischer Zucker (Überladen von Funktionen, Lambda-Funktionen etc.)
  - generische Funktionen und generische Klassen durch Templates
  - weniger *boilerplate code*
  - *zero-cost abstraction*

Warum C++ für HPC-Anwendungen?

> I have yet to see a program that can be written better in C than in C++. [. . .] I don't believe such a program could exist. By "better" I mean *smaller*, *more efficient*, or *more maintainable*.
>
> *Bjarne Stroustrup*

- Gründe für C++ (unvollständig & subjektiv):
  - striktes Typsystem & `const` *correctness*
  - Datenkapselung durch Klassen
  - automatisches Ressourcen-Management, *resource acquisition is initialization (RAII)*
  - syntaktischer Zucker (Überladen von Funktionen, Lambda-Funktionen etc.)
  - generische Funktionen und generische Klassen durch Templates
  - weniger *boilerplate code*
  - *zero-cost abstraction*

$\Rightarrow$ *erhöhte Produktivität & Wartbarkeit*

Hallo Parallelwelt! Message Passing mit MPI in C

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

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Hallo Parallelwelt! Message Passing mit MPI in C

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

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- mehrere Prozesse, ein Programm

Hallo Parallelwelt! Message Passing mit MPI in C

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

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- mehrere Prozesse, ein Programm
- Initialisierung/Finalisierung durch `MPI_Init`/`MPI_Finalize`

Hallo Parallelwelt! Message Passing mit MPI in C

```
#include <stdlib.h>
#include "mpi.h"

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- mehrere Prozesse, ein Programm
- Initialisierung/Finalisierung durch `MPI_Init`/`MPI_Finalize`
- jeder Prozess hat eindeutigen Rang ≥ 0, Code-Pfad abhängig vom Rang

Hallo Parallelwelt! Message Passing mit MPI in C

```
#include <stdlib.h>
#include "mpi.h"

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- mehrere Prozesse, ein Programm
- Initialisierung/Finalisierung durch `MPI_Init`/`MPI_Finalize`
- jeder Prozess hat eindeutigen Rang ≥ 0,
  Code-Pfad abhängig vom Rang
- Senden und Empfangen mit `MPI_Send`, `MPI_Recv`

Hallo Parallelwelt! Message Passing mit MPI in C

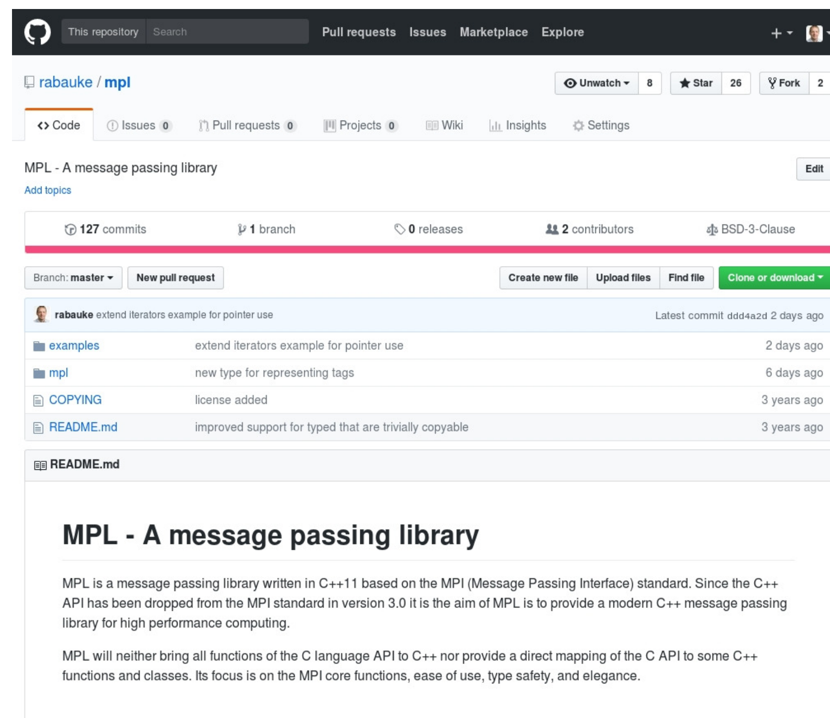
```
#include <stdlib.h>
#include "mpi.h"

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- mehrere Prozesse, ein Programm
- Initialisierung/Finalisierung durch `MPI_Init`/`MPI_Finalize`
- jeder Prozess hat eindeutigen Rang ≥ 0, Code-Pfad abhängig vom Rang
- Senden und Empfangen mit `MPI_Send`, `MPI_Recv`
- Nachrichten charakterisiert durch
  - Sendepuffer (`x`)
  - Nachrichtenlänge (`size`)
  - MPI-Datentyp (`MPI_DOUBLE`)
  - Rang des Kommunikationspartners (`0`, `1`)
  - Tag (`ping`, `pong`)
  - Kommunikator (`MPI_COMM_WORLD`)

Message Passing Library

Message Passing Library (MPL)

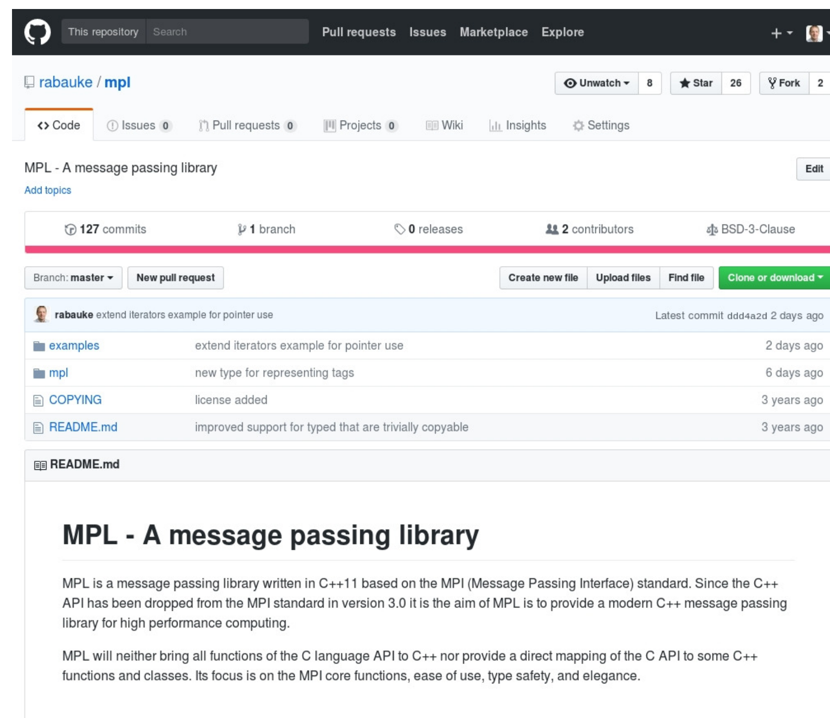
- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)



https://github.com/rabauke/mpl

Message Passing Library (MPL)

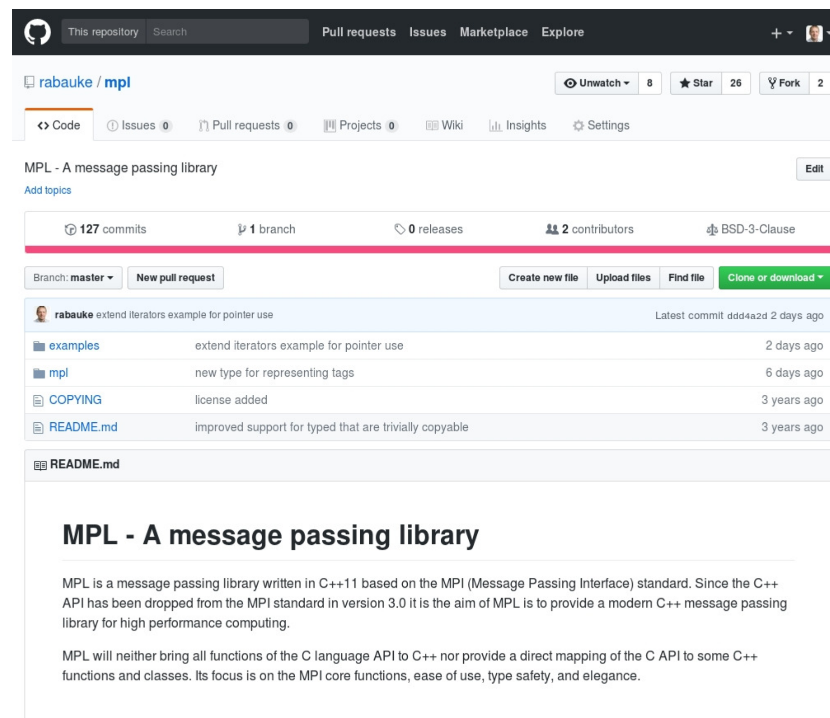
- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)
- C++-11, header-only



https://github.com/rabauke/mpl

Message Passing Library (MPL)

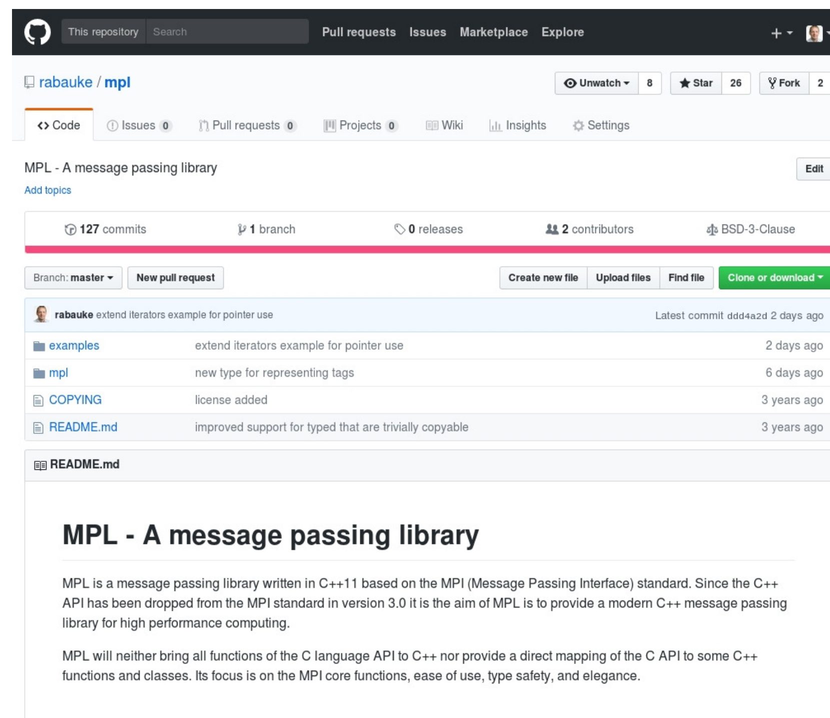
- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)
- C++-11, header-only
- basiert auf vielen C++-11 Features (Lambda-Funktionen, variadische Templates, `std::tuple` etc.)



https://github.com/rabauke/mpl

Message Passing Library (MPL)

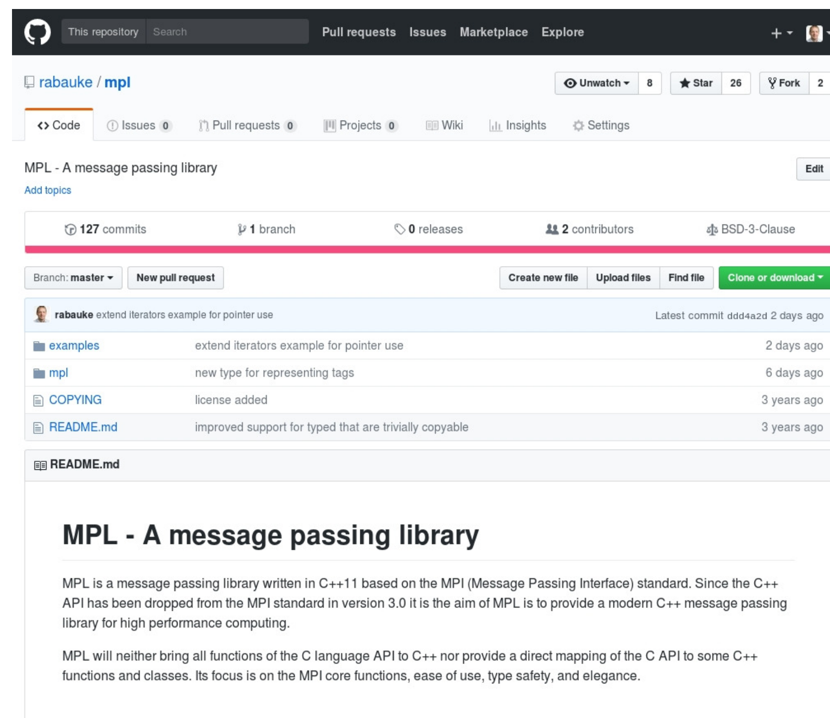
- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)
- C++-11, header-only
- basiert auf vielen C++-11 Features (Lambda-Funktionen, variadische Templates, `std::tuple` etc.)
- leichtgewichtige typsichere Abstraktionsschicht zwischen MPI und Anwendungscode



https://github.com/rabauke/mpl

Message Passing Library (MPL)

- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)
- C++-11, header-only
- basiert auf vielen C++-11 Features (Lambda-Funktionen, variadische Templates, `std::tuple` etc.)
- leichtgewichtige typsichere Abstraktionsschicht zwischen MPI und Anwendungscode
- Semantik der MPL-Funktionen orientiert an MPI-Funktionen;
  jedoch mit Abweichungen wo sinnvoll

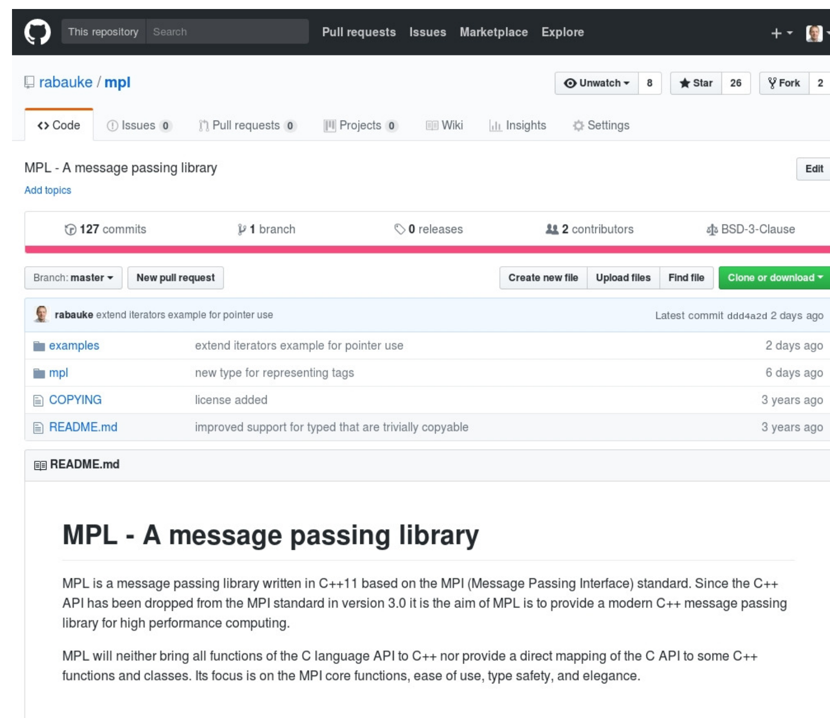


https://github.com/rabauke/mpl

Message Passing Library (MPL)

- Open-Source-Bibliothek auf der Basis von MPI (BSD-Lizenz)
- C++-11, header-only
- basiert auf vielen C++-11 Features (Lambda-Funktionen, variadische Templates, `std::tuple` etc.)
- leichtgewichtige typsichere Abstraktionsschicht zwischen MPI und Anwendungscode
- Semantik der MPL-Funktionen orientiert an MPI-Funktionen;
  jedoch mit Abweichungen wo sinnvoll
- deckt MPI-Funktionalität (MPI Version 3.1)
  - zum Nachrichtenaustausch (blocking & non-blocking),
  - zu abgeleiteten Datentypen,
  - zu Gruppen- und Kommunikator-Management,
  - zu Umgebungsmanagement,
  - zu Prozess-Topologien

  ab (kein I/O, dynamische Prozess-Erzeugung, einseitige Kommunikation)



https://github.com/rabauke/mpl

Hallo Parallelwelt! Message Passing mit MPL in C++

```
#include <stdlib.h>
#include "mpi.h"

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Hallo Parallelwelt! Message Passing mit MPL in C++

```
#include <stdlib.h>
#include "mpi.h"

#define SIZE 10

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int my_rank, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs!=2)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  const int size=SIZE;
  double x[SIZE]={ 0, 1, 2, 3, 4,
                   5, 6, 7, 8, 9 };
  enum { ping=1, pong=2 };
  if (my_rank==0) {
    MPI_Send(x, size, MPI_DOUBLE, 1, ping,
             MPI_COMM_WORLD);
    MPI_Recv(x, size, MPI_DOUBLE, 1, pong,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(x, size, MPI_DOUBLE, 0, ping,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(x, size, MPI_DOUBLE, 0, pong,
             MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &
    world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  double x[]
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  enum class tag : int { ping=1, pong=2 };
  if (world.rank()==0) {
    world.send(x, 1, tag::ping);
    world.recv(x, 1, tag::pong);
  } else {
    world.recv(x, 0, tag::ping);
    world.send(x, 0, tag::pong);
  }
  return EXIT_SUCCESS;
}
```

MPL-kompatible Datentypen

- Datentyp (`MPI_DOUBLE`) muss beim Senden und Empfangen *nicht* spezifiziert werden

MPL-kompatible Datentypen

- Datentyp (`MPI_DOUBLE`) muss beim Senden und Empfangen *nicht* spezifiziert werden
- Compile-Time-Type-Mapping *C++-Typ* ⇒ *MPI-Typ* für:
  - alle fundamentalen Typen (`int`, `double`, `bool`, etc.)
  - alle Enumerationstypen
  - alle komplexen Typen (`std::complex<double>` etc.)

MPL-kompatible Datentypen

- Datentyp (MPI_DOUBLE) muss beim Senden und Empfangen *nicht* spezifiziert werden
- Compile-Time-Type-Mapping *C++-Typ* ⇒ *MPI-Typ* für:
  - alle fundamentalen Typen (int, double, bool, etc.)
  - alle Enumerationstypen
  - alle komplexen Typen (std::complex<double> etc.)
- Compile-Time-Type-Mapping *C++-Typ* ⇒ *Code* zur Erzeugung von MPI-Typ zur Laufzeit für:
  - C-Arrays fester Größe
  - Template-Klassen std::array, std::pair und std::tuple
  - Typen der Elemente müssen für Kommunikation geeignet sein
    - Regel rekursiv anwendbar

```
std::tuple<std::array<std::pair<int, double>, 8>, int, bool>
```

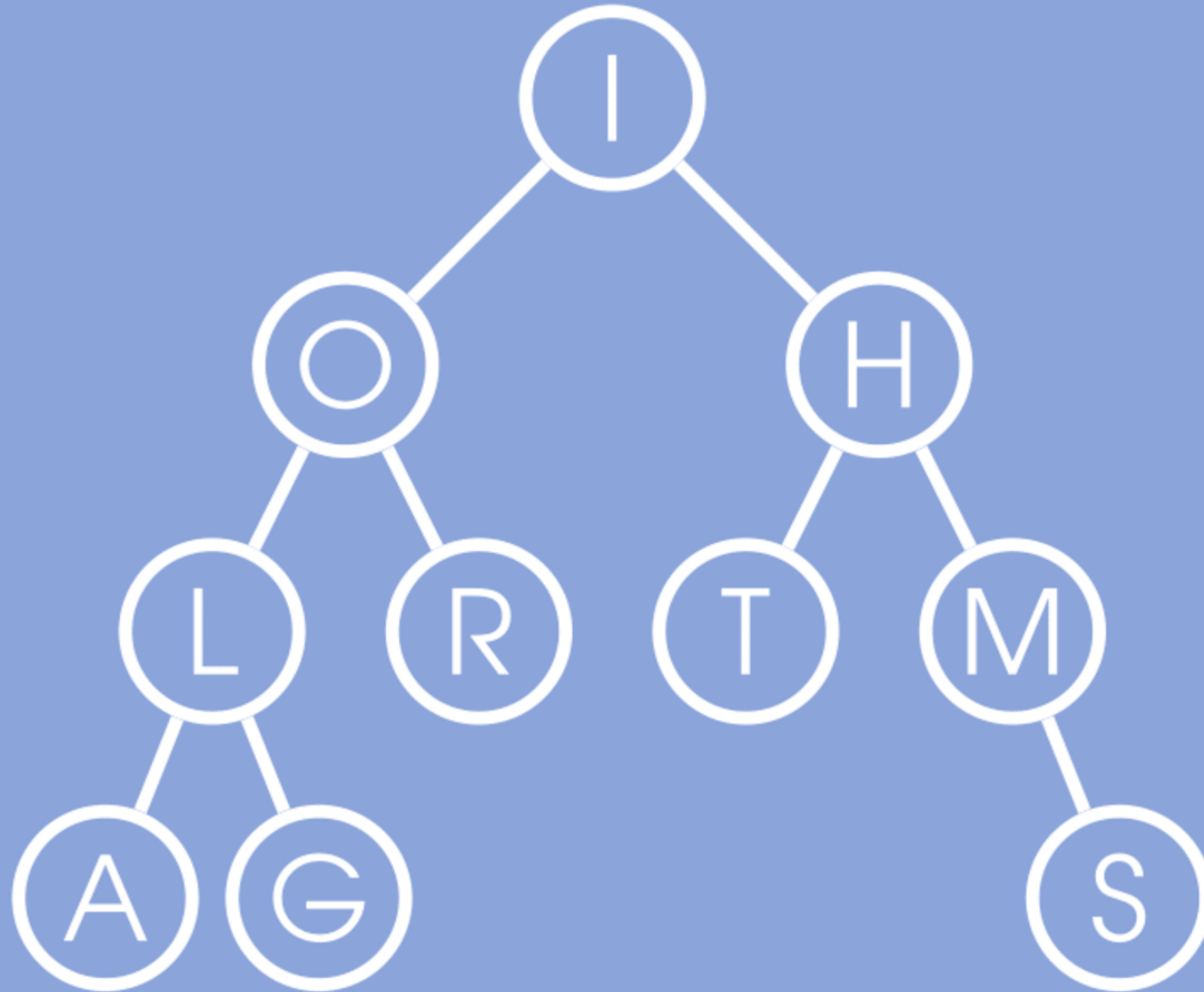
  - trivial (bit-weise) kopierbare Typen (falls Makro MPL_HOMOGENEOUS definiert)

MPL-kompatible Datentypen

- Datentyp (`MPI_DOUBLE`) muss beim Senden und Empfangen *nicht* spezifiziert werden
- Compile-Time-Type-Mapping *C++-Typ* ⇒ *MPI-Typ* für:
  - alle fundamentalen Typen (`int`, `double`, `bool`, etc.)
  - alle Enumerationstypen
  - alle komplexen Typen (`std::complex<double>` etc.)
- Compile-Time-Type-Mapping *C++-Typ* ⇒ *Code* zur Erzeugung von MPI-Typ zur Laufzeit für:
  - C-Arrays fester Größe
  - Template-Klassen `std::array`, `std::pair` und `std::tuple`
  - Typen der Elemente müssen für Kommunikation geeignet sein
    - Regel rekursiv anwendbar

      ```
      std::tuple<std::array<std::pair<int, double>, 8>, int, bool>
      ```
  - trivial (bit-weise) kopierbare Typen (falls Makro `MPL_HOMOGENEOUS` definiert)
- MPI-Typen werden *bei Bedarf einmal* erzeugt und bei Programmende zerstört (Meyer's singleton)

Nutzerdefinierte Datentypen



Nutzerdefinierte Datentypen

- MPI/MPL benötigt Information über interne Datenrepräsentation nutzerdefinierter Datentypen/Klassen
- im Allgemeinen nicht zugänglich (*Datenkapselung* in OO-Sprachen)

Nutzerdefinierte Datentypen

- MPI/MPL benötigt Information über interne Datenrepräsentation nutzerdefinierter Datentypen/Klassen
- im Allgemeinen nicht zugänglich (*Datenkapselung* in OO-Sprachen)
- Lösung: Reflexion/Introspektion
  (kein Sprach-Feature von C++-17, teilweise realisierbar für Template-Klassen)

Nutzerdefinierte Datentypen

- MPI/MPL benötigt Information über interne Datenrepräsentation nutzerdefinierter Datentypen/Klassen
- im Allgemeinen nicht zugänglich (*Datenkapselung* in OO-Sprachen)
- Lösung: Reflexion/Introspektion
  (kein Sprach-Feature von C++-17, teilweise realisierbar für Template-Klassen)
- Strukturen ausschließlich mit *öffentlichen* Datenelementen:
  Makro `MPL_REFLECTION` ⇒ spezialisiert Template-Klasse `mpl::struct_builder`

```
struct structure {
  double d=0;
  int i[9]={0, 0, 0, 0, 0, 0, 0, 0, 0};
  structure()=default;
};

MPL_REFLECTION(structure, d, i)

struct structure2 {
  double d=0;
  structure str;
  structure2()=default;
};

MPL_REFLECTION(structure2, d, str)
```

Nutzerdefinierte Datentypen

- MPI/MPL benötigt Information über interne Datenrepräsentation nutzerdefinierter Datentypen
- im Allgemeinen nicht zugänglich (*Datenkapselung* in OO-Sprachen)
- Lösung: Reflexion/Introspektion
  (kein Sprach-Feature von C++-17, teilweise realisierbar für Template-Klassen)
- Strukturen ausschließlich mit *öffentlichen* Datenelementen:
  Makro `MPL_REFLECTION` ⇒ spezialisiert Template-Klasse `mpl::struct_builder`
- Strukturen mit *privaten* Datenelementen:
  friend-Deklaration von `mpl::struct_builder` & Makro `MPL_REFLECTION`

```
struct structure {
private:
  double d=0;
  std::array<int, 9> i={ 0, 0, 0, 0, 0, 0, 0, 0, 0 };
public:
  structure()=default;
  structure(double d, const std::array<int, 9> &i) : d(d), i(i) {
  }
  friend class mpl::struct_builder<structure>;
};

MPL_REFLECTION(structure, d, i)
```

MPI versus MPL

- Senden und Empfangen in MPI:

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

MPI versus MPL

- Senden und Empfangen in MPI:

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

- MPI-Datentyp codiert Information über
  - grundlegenden C-Datentyp (`double` etc.)
  - Speicherlayout bei Nachrichten mit mehreren Elementen (Vektor mit N Elementen, obere Dreiecksmatrix etc.)

MPI versus MPL

- Senden und Empfangen in MPI:

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

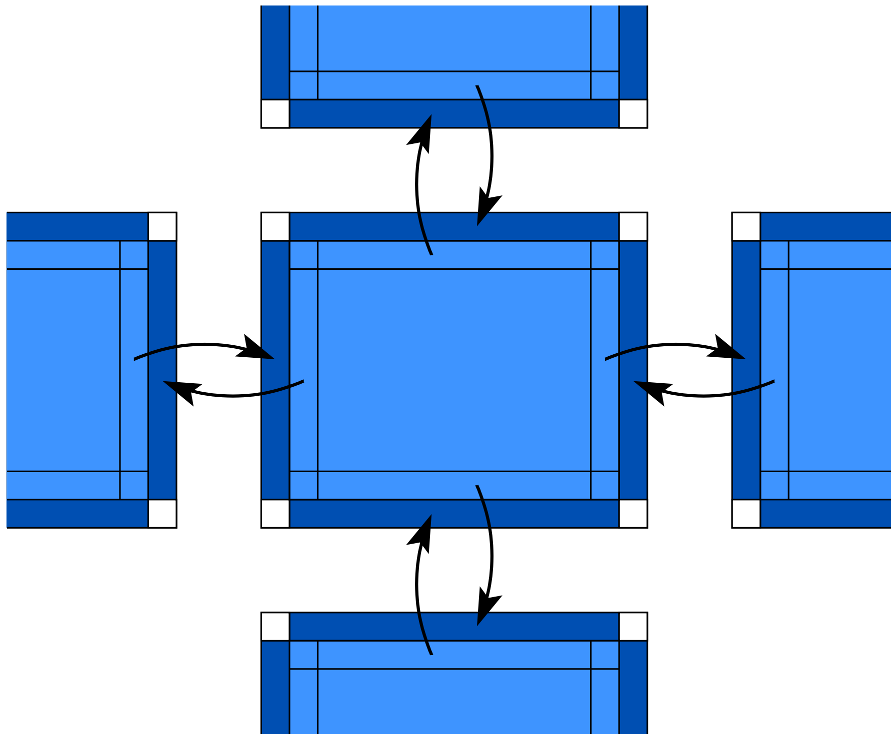
- MPI-Datentyp codiert Information über
  - grundlegenden C-Datentyp (`double` etc.)
  - Speicherlayout bei Nachrichten mit mehreren Elementen (Vektor mit N Elementen, obere Dreiecksmatrix etc.)
- Senden und Empfangen in MPL:

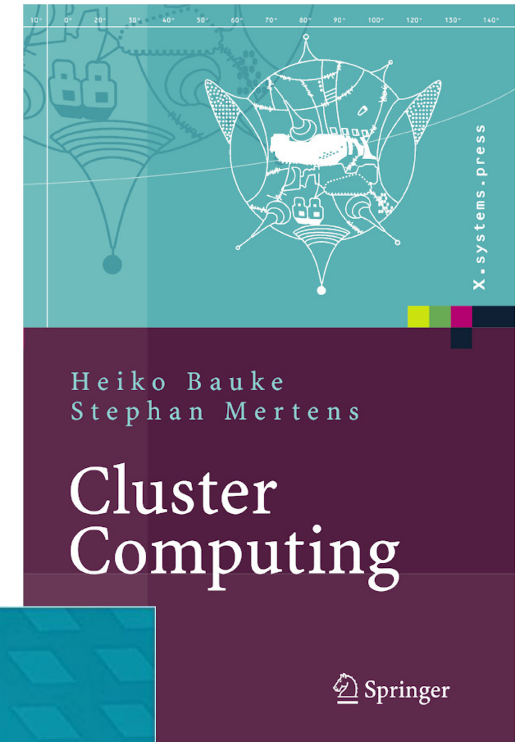
```
template<typename T>
void mpl::communicator::send(const T &data, int dest, mpl::tag t=mpl::tag(0)) const
template<typename T>
mpl::status mpl::communicator::recv(T &data, int source, mpl::tag t=mpl::tag(0)) const
```

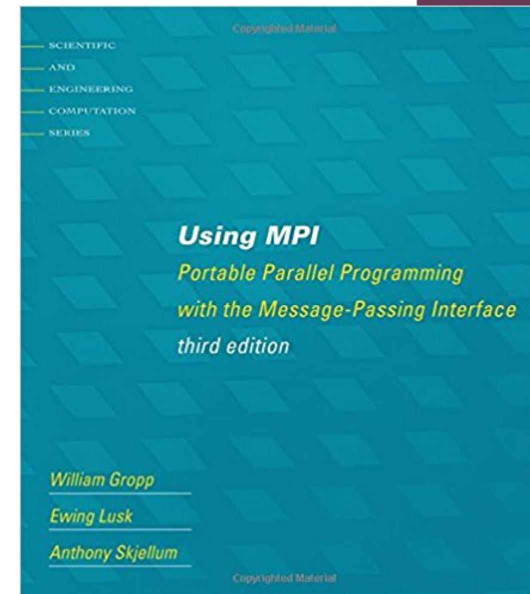
  - C++-Datentyp der Nachricht vom Compiler ermittelt (*Typsicherheit!*)
  - Tag durch eigenen Typ repräsentiert (*Typsicherheit!*)
  - kein `count`-Argument
  - Nachricht immer genau *ein* Element (`double`, `int [10]` etc.)

Layouts

- MPI-Datentypen repräsentieren Datentypen und *Speicherlayout* von Nachrichten ⇒ essentiell für klar strukturierte MPI-Programme







Layouts

- MPI-Datentypen repräsentieren Datentypen und *Speicherlayout* von Nachrichten ⇒ essentiell für klar strukturierte MPI-Programme
- `count`-Argument > 1 kann in MPI-Datentyp absorbiert werden

Layouts

- MPI-Datentypen repräsentieren Datentypen und *Speicherlayout* von Nachrichten ⇒ essentiell für klar strukturierte MPI-Programme
- `count`-Argument > 1 kann in MPI-Datentyp absorbiert werden
- MPL repräsentiert Speicherlayout durch Layouts

Layouts

- MPI-Datentypen repräsentieren Datentypen und *Speicherlayout* von Nachrichten ⇒ essentiell für klar strukturierte MPI-Programme
- `count`-Argument > 1 kann in MPI-Datentyp absorbiert werden
- MPL repräsentiert Speicherlayout durch Layouts

```
template<typename T>
void mpl::communicator::send(const T *data, const mpl::layout<T> &l, int dest,
                             mpl::tag t=mpl::tag(0)) const
template<typename T>
mpl::status mpl::communicator::recv(T *data, const mpl::layout<T> &l, int source,
                                    mpl::tag t=mpl::tag(0)) const
```

Layouts

- MPI-Datentypen repräsentieren Datentypen und *Speicherlayout* von Nachrichten ⇒ essentiell für klar strukturierte MPI-Programme
- `count`-Argument > 1 kann in MPI-Datentyp absorbiert werden
- MPL repräsentiert Speicherlayout durch Layouts

```
template<typename T>
void mpl::communicator::send(const T *data, const mpl::layout<T> &l, int dest,
                             mpl::tag t=mpl::tag(0)) const
template<typename T>
mpl::status mpl::communicator::recv(T *data, const mpl::layout<T> &l, int source,
                                    mpl::tag t=mpl::tag(0)) const
```

- Layout-Klassen

```
namespace mpl {
  template<typename T> class layout;                  // Basisklasse
  template<typename T> class null_layout;             // MPI_DATATYPE_NULL
  template<typename T> class empty_layout;            // leere Nachricht
  template<typename T> class contiguous_layout;       // MPI_Type_contiguous
  template<typename T> class vector_layout;           // MPI_Type_contiguous
  template<typename T> class strided_vector_layout;   // MPI_Type_vector
  template<typename T> class indexed_layout;          // MPI_Type_indexed
  template<typename T> class hindexed_layout;         // MPI_Type_create_hindexed
  template<typename T> class indexed_block_layout;    // MPI_Type_create_indexed_block
  template<typename T> class hindexed_block_layout;   // MPI_Type_create_hindexed_block
  template<typename T> class iterator_layout;         // MPI_Type_create_hindexed_block
  template<typename T> class subarray_layout;         // MPI_Type_create_subarray
  class heterogeneous_layout;                         // MPI_Type_create_struct
}
```

Layouts in Aktion

zusammenhängender Speicherbereich mit Vektor-Layout

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  std::vector<double> x(10);
  mpl::vector_layout<double> l(x.size());
  if (world.rank()==0) {
    std::iota(begin(x), end(x), 0);
    world.send(x.data(), l, 1);   // sende 10 double
  } else {
    world.recv(x.data(), l, 0);   // empfange 10 double
    std::for_each(begin(x), end(x), [](auto x){ std::cout << x << '\n'; });
  }
  return EXIT_SUCCESS;
}
```

Layouts in Aktion

mehrere Datenelemente unterschiedlichen Typs in einer Nachricht

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  double r;
  int i;
  std::vector<double> x(10);
  mpl::vector_layout<double> lx(x.size());
  mpl::heterogeneous_layout hl(r, i, mpl::make_absolute(x.data(), lx));
  if (world.rank()==0) {
    r=3.14;
    i=42;
    std::iota(begin(x), end(x), 0);
    world.send(mpl::absolute, hl, 1);   // sende r, i & x
  } else {
    world.recv(mpl::absolute, hl, 0);   // empfange r, i & x
    std::cout << r << '\n' << i << '\n';
    std::for_each(begin(x), end(x), [](auto x){ std::cout << x << '\n'; });
  }
  return EXIT_SUCCESS;
}
```

Layouts in Aktion

unzusammenhängender Speicherbereich mit Index-Layout

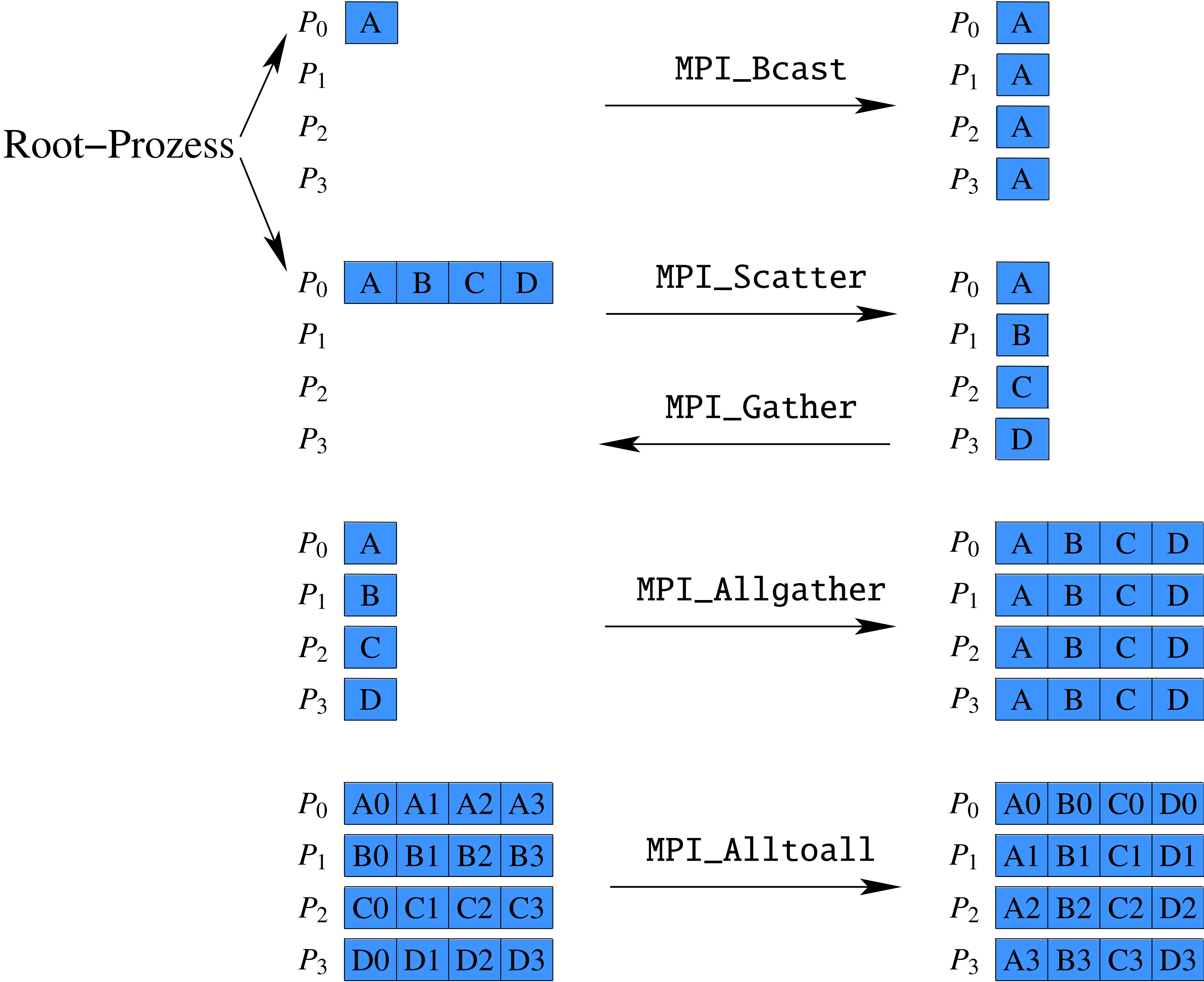
```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <array>
#include <numeric>
#include <algorithm>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  if (world.rank()==0) {
    mpl::indexed_layout<double> l({
      {2, 0}, // 2 Elemente ab Position 0
      {1, 4}  // 1 Element ab Position 4
    });
    std::vector<double> x(100);
    std::iota(begin(x), end(x), 0);
    world.send(x.data(), l, 1);  // sende 3 double
  } else {
    std::array<double, 3> x;
    world.recv(x, 0);    // empfange 3 double
    // Ausgabe: 0 1 4
    std::for_each(begin(x), end(x), [](auto x){ std::cout << x << '\n'; });
  }
  return EXIT_SUCCESS;
}
```

Layouts in Aktion

Vektor aus unzusammenhängendem Speicherbereich mit Index- & Vektor-Layout

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <array>
#include <numeric>
#include <algorithm>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  mpl::indexed_layout<double> l({
      {2, 0}, //  2 Elemente ab Position 0
      {1, 4}  //  1 Element ab Position 4
  });
  mpl::vector_layout<double> lv(20, l);   //  Vektorlayout aus indiziertem Layout
  std::vector<double> x(100, 0);   //  100 Elemente mit 0 initialisiert
  if (world.rank()==0) {
    std::iota(begin(x), end(x), 1);
    world.send(x.data(), lv, 1);   //  sende ausgewählte Elemente von x
  } else {
    world.recv(x.data(), lv, 0);   //  empfange ausgewählte Elemente von x
    //  Ausgabe: 1 2 0 0 5  6 7 0 0 10  11 12 0 0 15 ...
    std::for_each(begin(x), end(x), [](auto x){ std::cout << x << '\n'; });
  }
  return EXIT_SUCCESS;
}
```

Kollektive Kommunikation



Kollektive Kommunikation

- kollektive Operationen überladen, Beispiel Scatter

```
template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, T &recvdata) const

template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, const mpl::layout<T> &sendl,
                                T *recvdata, const mpl::layout<T> &recvl) const
```

Kollektive Kommunikation

- kollektive Operationen überladen, Beispiel Scatter

```
template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, T &recvdata) const

template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, const mpl::layout<T> &sendl,
                                T *recvdata, const mpl::layout<T> &recvl) const
```

- nicht-signifikante Parameter auf Seiten der Nicht-Root-Prozesse nicht benutzt

Kollektive Kommunikation

- kollektive Operationen überladen, Beispiel Scatter

```
template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, T &recvdata) const

template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, const mpl::layout<T> &sendl,
                                T *recvdata, const mpl::layout<T> &recvl) const
```

- nicht-signifikante Parameter auf Seiten der Nicht-Root-Prozesse nicht benutzt
- Nicht-Root-Prozesse benutzen gegebenenfalls `empty_layout` und Null-Pointer

Kollektive Kommunikation

- kollektive Operationen überladen, Beispiel Scatter

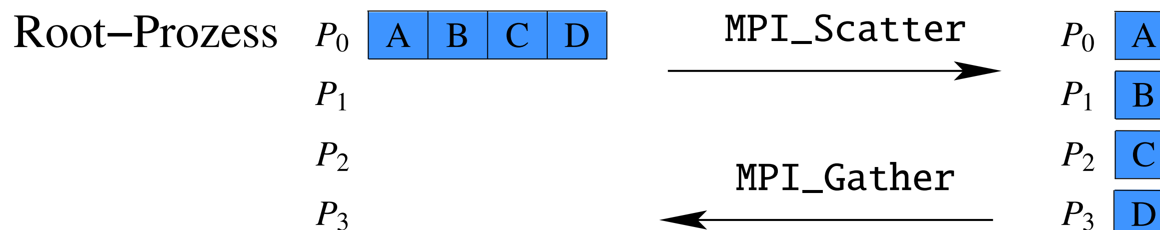
```
template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, T &recvdata) const

template<typename T>
void mpl::communicator::scatter(int root, const T *senddata, const mpl::layout<T> &sendl,
                                T *recvdata, const mpl::layout<T> &recvl) const
```

- nicht-signifikante Parameter auf Seiten der Nicht-Root-Prozesse nicht benutzt
- Nicht-Root-Prozesse benutzen gegebenenfalls `empty_layout` und Null-Pointer
- … oder überladene Varianten für Nicht-Root-Prozesse

```
//  Varianten ausschließlich für Prozesse, die nicht selbst Root
template<typename T>
void mpl::communicator::scatter(int root, T &recvdata) const

template<typename T>
void mpl::communicator::scatter(int root, T *recvdata, const layout<T> &recvl) const
```

Kollektive Kommunikation

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <numeric>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world=mpl::environment::comm_world();
  if (world.rank()==0) {
    std::vector<int> v(world.size());
    std::iota(begin(v), end(v), 1);
    int x;
    world.scatter(0, v.data(), x);   // verteile Elemente von v
    x*=2;
    world.gather(0, x, v.data());    // sammle Elemente von v
    for (int i=0; i<world.size(); ++i)
      std::cout << "erhielt " << v[i] << " von Rang " << i << '\n';
  } else {
    int x;
    world.scatter(0, x);   // Nicht-Root empfängt nur
    x*=2;
    world.gather(0, x);    // Nicht-Root sendet nur
  }
  return EXIT_SUCCESS;
}
```



Reduktionsoperationen

- Reduktionsoperationen
  - Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei Root
  - vollständige Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei allen Prozessen
  - Reduktion mit anschließendem Scatter
  - Präfixreduzierung (scan & exclusive scan)

Reduktionsoperationen

- Reduktionsoperationen
  - Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei Root
  - vollständige Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei allen Prozessen
  - Reduktion mit anschließendem Scatter
  - Präfixreduzierung (scan & exclusive scan)

```
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world=mpl::environment::comm_world();
  double x=world.rank()+1;
  world.allreduce(mpl::plus<double>(), x);
  std::cout << "Summe " << x << '\n';
  return EXIT_SUCCESS;
}
```

Reduktionsoperationen

- Reduktionsoperationen
  - Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei Root
  - vollständige Reduktion: Summe, Produkt etc. über alle Prozesse
    $\Rightarrow$ Ergebnis bei allen Prozessen
  - Reduktion mit anschließendem Scatter
  - Präfixreduzierung (scan & exclusive scan)

```
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world=mpl::environment::comm_world();
  double x=world.rank()+1;
  world.allreduce(mpl::plus<double>(), x);
  std::cout << "Summe " << x << '\n';
  return EXIT_SUCCESS;
}
```

- Reduktionsoperationen für Layouts überladen

Reduktionsoperatoren

- MPL-Reduktionsoperatoren
  - `max`
  - `min`
  - `plus`
  - `multiplies`
  - `logical_and`
  - `logical_or`
  - `logical_xor`
  - `bit_and`
  - `bit_or`
  - `bit_xor`

Reduktionsoperatoren

- MPL-Reduktionsoperatoren
  - `max`
  - `min`
  - `plus`
  - `multiplies`
  - `logical_and`
  - `logical_or`
  - `logical_xor`
  - `bit_and`
  - `bit_or`
  - `bit_xor`
- Signatur von Reduktionsoperatoren: `T op(T, T)`

Reduktionsoperatoren

- MPL-Reduktionsoperatoren
  - `max`
  - `min`
  - `plus`
  - `multiplies`
  - `logical_and`
  - `logical_or`
  - `logical_xor`
  - `bit_and`
  - `bit_or`
  - `bit_xor`
- Signatur von Reduktionsoperatoren: `T op(T, T)`
- wirken elementweise bei Nachrichten mit mehreren Elementen

Reduktionsoperatoren

- MPL-Reduktionsoperatoren
  - max
  - min
  - plus
  - multiplies
  - logical_and
  - logical_or
  - logical_xor
  - bit_and
  - bit_or
  - bit_xor
- Signatur von Reduktionsoperatoren: `T op(T, T)`
- wirken elementweise bei Nachrichten mit mehreren Elementen
- neue Reduktionsoperatoren über Lambda-Funktionen realisierbar

```
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

using pair=std::pair<double, double>;

int main() {
  const mpl::communicator &world=mpl::environment::comm_world();
  pair x(world.rank()+1, world.rank()+1);
  world.allreduce([](pair a, pair b){
      return pair(a.first+b.first, a.second*b.second);
    }, x);
  std::cout << "Summe " << x.first << '\n';
  std::cout << "Produkt " << x.second << '\n';
  return EXIT_SUCCESS;
}
```

Nicht-blockierender Nachrichtenaustausch

- nicht-blockierender Nachrichtenaustausch:
  - zur Vermeidung von Deadlocks
  - steigert (potentiell) Performance
  - zeitliche Überlagerung von Kommunikation und Berechnungen

Nicht-blockierender Nachrichtenaustausch

- nicht-blockierender Nachrichtenaustausch:
  - zur Vermeidung von Deadlocks
  - steigert (potentiell) Performance
  - zeitliche Überlagerung von Kommunikation und Berechnungen
- Sende- und Empfangsfunktionen leiten Nachrichtenaustausch nur ein

Nicht-blockierender Nachrichtenaustausch

- nicht-blockierender Nachrichtenaustausch:
  - zur Vermeidung von Deadlocks
  - steigert (potentiell) Performance
  - zeitliche Überlagerung von Kommunikation und Berechnungen
- Sende- und Empfangsfunktionen leiten Nachrichtenaustausch nur ein
- Status des Nachrichtenaustauschs über Request-Objekt abfragbar

Nicht-blockierender Nachrichtenaustausch

- nicht-blockierender Nachrichtenaustausch:
  - zur Vermeidung von Deadlocks
  - steigert (potentiell) Performance
  - zeitliche Überlagerung von Kommunikation und Berechnungen
- Sende- und Empfangsfunktionen leiten Nachrichtenaustausch nur ein
- Status des Nachrichtenaustauschs über Request-Objekt abfragbar

```
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  double x=world.rank()+1, y;
  if (world.rank()==0) {
    mpl::irequest request1(world.isend(x, 1));   // starte Senden
    mpl::irequest request2(world.irecv(y, 1));   // starte Empfangen
    request1.wait();   // warete auf Abschluss der Sendeoperation
    request2.wait();   // warete auf Abschluss der Empfangsoperation
  } else {
    mpl::irequest_pool requests;   // Pool von Requests
    requests.push(world.isend(x, 0));   // starte Senden und füge Request in Pool
    requests.push(world.irecv(y, 0));   // starte Empfangen und füge Request in Pool
    requests.waitall();   // warete auf Abschluss aller Kommunikationsoperationen
  }
  std::cout << "Rang " << world.rank() << " empfing " << y << '\n';
  return EXIT_SUCCESS;
}
```

Nicht-blockierender Nachrichtenaustausch

- nicht-blockierender Nachrichtenaustausch:
  - zur Vermeidung von Deadlocks
  - steigert (potentiell) Performance
  - zeitliche Überlagerung von Kommunikation und Berechnungen
- Sende- und Empfangsfunktionen leiten Nachrichtenaustausch nur ein
- Status des Nachrichtenaustauschs über Request-Objekt abfragbar

```cpp
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world(mpl::environment::comm_world());
  if (world.size()!=2)
    world.abort(EXIT_FAILURE);
  double x=world.rank()+1, y;
  if (world.rank()==0) {
    mpl::irequest request1(world.isend(x, 1));  // starte Senden
    mpl::irequest request2(world.irecv(y, 1));  // starte Empfangen
    request1.wait();  // warete auf Abschluss der Sendeoperation
    request2.wait();  // warete auf Abschluss der Empfangsoperation
  } else {
    mpl::irequest_pool requests;  // Pool von Requests
    requests.push(world.isend(x, 0));  // starte Senden und füge Request in Pool
    requests.push(world.irecv(y, 0));  // starte Empfangen und füge Request in Pool
    while (not requests.testall()) {  /* rechne */  }
  }
  std::cout << "Rang " << world.rank() << " empfing " << y << '\n';
  return EXIT_SUCCESS;
}
```

Ausblick auf weitere Funktionen

- weitere Funktionen:

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren

```
#include <cstdlib>
#include <vector>
#include <set>
#include <iostream>
#include <numeric>
#include <algorithm>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &world=mpl::environment::comm_world();
  if (world.size()<2)
    world.abort(EXIT_FAILURE);
  const int N=10;
  if (world.rank()==0) {
    std::set<double> s;
    for (int i=0; i<N; ++i)  // fülle Set-Container
      s.insert(i);
    world.send(begin(s), end(s), 1);  // sende Set-Container
  }
  if (world.rank()==1) {
    std::vector<double> v(N);
    world.recv(begin(v), end(v), 0);  // empfange Daten in STL-Vektor
  }
  return EXIT_SUCCESS;
}
```

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren
  - kollektive Operationen mit Nachrichten variabler Größe

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren
  - kollektive Operationen mit Nachrichten variabler Größe
  - Probe & Cancel

Ausblick auf weitere Funktionen

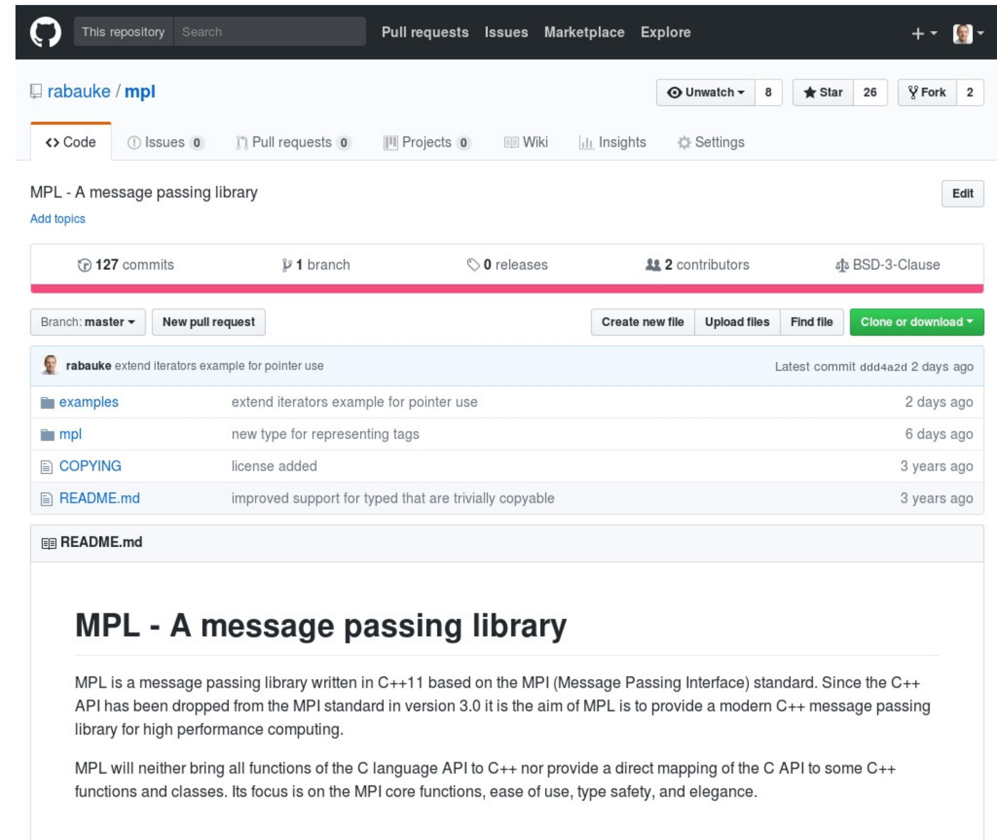
- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren
  - kollektive Operationen mit Nachrichten variabler Größe
  - Probe & Cancel
  - Kommunikator-Management / kartesische Kommunikatoren

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren
  - kollektive Operationen mit Nachrichten variabler Größe
  - Probe & Cancel
  - Kommunikator-Management / kartesische Kommunikatoren
  - Klasse `distributed_grid` für Gebietszerlegung in d-Dimensionen

Ausblick auf weitere Funktionen

- weitere Funktionen:
  - MPL unterstützt alle MPI-Sendemodi (standard, buffered, synchronous, ready) jeweils blockierend & nicht-blockierend
  - Senden / Empfangen von STL-Containern via Iteratoren
  - kollektive Operationen mit Nachrichten variabler Größe
  - Probe & Cancel
  - Kommunikator-Management / kartesische Kommunikatoren
  - Klasse `distributed_grid` für Gebietszerlegung in d-Dimensionen
- siehe Beispiele auf Github https://github.com/rabauke/mpl



Zusammenfassung

Zusammenfassung

- HPC-Anwendungen erfordern effiziente ausdrucksstarke Sprachen!
  $\Rightarrow$ C++-11 / C++-17

Zusammenfassung

- HPC-Anwendungen erfordern effiziente ausdrucksstarke Sprachen!
  $\Rightarrow$ C++-11 / C++-17
- Message Passing Library (MPL) als Basis moderner HPC-Anwendungen

Zusammenfassung

- HPC-Anwendungen erfordern effiziente ausdrucksstarke Sprachen!
  ⇒ C++-11 / C++-17
- Message Passing Library (MPL) als Basis moderner HPC-Anwendungen
- Vorteile der MPL gegenüber MPI:
  - erhöhte Typsicherheit (keine void-Pointer)
  - Compile-Time- statt Run-Time-Fehler
  - automatisches Ressourcen-Management durch RAII (automatische Freigabe von MPI-Handles)
  - automatisches Bauen von MPI-Datentypen im Hintergrund
  - praktisch kein Overhead im Vergleich zu MPI durch Inline-Funktionen und Template-Metaprogrammierung
  - teils erhebliche Reduktion von boilerplate code

Zusammenfassung

```c
typedef struct vector {
  double *data;
  size_t N;
} vector;

void fill_random(vector v) {
  for (size_t i=0; i<v.N; ++i)
    v.data[i]=(double) rand()/(RAND_MAX+1.);
}

static int cmp_double(const void *p1_, const void *p2_) {
  const double const *p1=p1_, *p2=p2_;
  return (*p1==*p2) ? 0 : (*p1<*p2 ? -1 : 1);
}

double *partition(double *first, double *last, double pivot) {
  for (; first!=last; ++first)
    if (!((*first)<pivot))
      break;
  if (first==last)
    return first;
  for (double *i=first+1; i!=last; ++i) {
    if ((*i)<pivot) {
      double temp=*i;
      *i=*first;
      *first=temp;
      ++first;
    }
  }
  return first;
}

vector parallel_sort(vector v) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  double *local_pivots=malloc(size*sizeof(*local_pivots));
  if (local_pivots==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  double *pivots=malloc(size*(size+1)*sizeof(*pivots));
  if (pivots==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  for (int i=0; i<size-1; ++i)
    local_pivots[i]=v.data[(size_t) (v.N*(double) rand()/(RAND_MAX+1.))];
  MPI_Allgather(local_pivots, size-1, MPI_DOUBLE,
                pivots, size-1, MPI_DOUBLE,
                MPI_COMM_WORLD);
  qsort(pivots, size*(size-1), sizeof(double), cmp_double);
  for (size_t i=1; i<size; ++i)
    local_pivots[i-1]=pivots[i*(size-1)];
  double **pivot_pos=malloc((size+1)*sizeof(*pivot_pos));
  if (pivot_pos==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  pivot_pos[0]=v.data;
  for (size_t i=0; i<size-1; ++i)
    pivot_pos[i+1]=partition(pivot_pos[i], v.data+v.N, local_pivots[i]);
  pivot_pos[size]=v.data+v.N;
  int *local_block_sizes=malloc(size*sizeof(*local_block_sizes));
  if (local_block_sizes==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  int *block_sizes=malloc(size*size*sizeof(*block_sizes));
  if (block_sizes==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  for (size_t i=0; i<size; ++i)
    local_block_sizes[i]=pivot_pos[i+1]-pivot_pos[i];
  MPI_Allgather(local_block_sizes, size, MPI_INT,
                block_sizes, size, MPI_INT,
                MPI_COMM_WORLD);
  int send_pos=0, recv_pos=0;
  int sendcounts[size], sdispls[size], recvcounts[size], rdispls[size];
  for (size_t i=0; i<size; ++i) {
    sendcounts[i]=block_sizes[rank*size+i];
    sdispls[i]=send_pos;
    send_pos+=block_sizes[rank*size+i];
    recvcounts[i]=block_sizes[rank+size*i];
    rdispls[i]=recv_pos;
    recv_pos+=block_sizes[rank+size*i];
  }
  double *v2=malloc(recv_pos*sizeof(*v2));
  if (v2==NULL)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  MPI_Alltoallv(v.data, sendcounts, sdispls, MPI_DOUBLE,
                v2, recvcounts, rdispls, MPI_DOUBLE,
                MPI_COMM_WORLD);
  qsort(v2, recv_pos, sizeof(double), cmp_double);
  free(v.data);
  free(block_sizes);
  free(local_block_sizes);
  free(pivot_pos);
  free(pivots);
  free(local_pivots);
  return (vector) { v2, recv_pos };
}
```

```cpp
template<typename T>
void parallel_sort(std::vector<T> &v) {
  auto comm_world{ mpl::environment::comm_world() };
  const int rank{ comm_world.rank() };
  const int size{ comm_world.size() };
  std::vector<T> local_pivots, pivots(size*(size-1));
  std::sample(begin(v), end(v), std::back_inserter(local_pivots), size-1, mt);
  comm_world.allgather(local_pivots.data(), mpl::vector_layout<T>(size-1),
                       pivots.data(), mpl::vector_layout<T>(size-1));
  std::sort(begin(pivots), end(pivots));
  local_pivots.resize(0);
  for (std::size_t i=1; i<size; ++i)
    local_pivots.push_back(pivots[i*(size-1)]);
  swap(local_pivots, pivots);
  std::vector<typename std::vector<T>::iterator> pivot_pos;
  pivot_pos.push_back(begin(v));
  for (T p : pivots)
    pivot_pos.push_back(std::partition(pivot_pos.back(), end(v), [p](T x) { return x<p; }));
  pivot_pos.push_back(end(v));
  std::vector<int> local_block_sizes, block_sizes(size*size);
  for (std::size_t i=0; i<pivot_pos.size()-1; ++i)
    local_block_sizes.push_back(static_cast<int>(std::distance(pivot_pos[i], pivot_pos[i+1])));
  comm_world.allgather(local_block_sizes.data(), mpl::vector_layout<int>(size),
                       block_sizes.data(), mpl::vector_layout<int>(size));
  mpl::layouts<T> send_layouts, recv_layouts;
  int send_pos{ 0 }, recv_pos{ 0 };
  for (int i=0; i<size; ++i) {
    send_layouts.push_back(mpl::indexed_layout<T>({{ block_sizes[rank*size+i], send_pos }}));
    send_pos+=block_sizes[rank*size+i];
    recv_layouts.push_back(mpl::indexed_layout<T>({{ block_sizes[rank+size*i], recv_pos }}));
    recv_pos+=block_sizes[rank+size*i];
  }
  std::vector<T> v2(recv_pos);
  comm_world.alltoallv(v.data(), send_layouts, v2.data(), recv_layouts);
  std::sort(begin(v2), end(v2));
  swap(v, v2);
}
```

paralleler Sortieralgorithmus

C & MPI vs. C++ & MPL

93 lines of code vs. 37 lines of code

Zusammenfassung

- HPC-Anwendungen erfordern effiziente ausdrucksstarke Sprachen!
  ⇒ C++-11 / C++-17
- Message Passing Library (MPL) als Basis moderner HPC-Anwendungen
- Vorteile der MPL gegenüber MPI:
  - erhöhte Typsicherheit (keine void-Pointer)
  - Compile-Time- statt Run-Time-Fehler
  - automatisches Ressourcen-Management durch RAII (automatische Freigabe von MPI-Handles)
  - automatisches Bauen von MPI-Datentypen im Hintergrund
  - praktisch kein Overhead im Vergleich zu MPI durch Inline-Funktionen und Template-Metaprogrammierung
  - teils erhebliche Reduktion von boilerplate code
- weiterhin bestehende Limitationen von MPI/MPL:
  - Empfänger muss Nachrichtengröße im Voraus kennen und Speicher bereitstellen
  - mangelnde Möglichkeiten zu Reflexion/Introspektion in C++
  - Datentypen mit Elementen dynamischer Größe schlecht unterstützt
  - kaum Toleranz gegenüber Run-Time-Fehler
  - 32-Bit-Integer-API-Desaster

Zusammenfassung

- HPC-Anwendungen erfordern effiziente ausdrucksstarke Sprachen!
  ⇒ C++-11 / C++-17
- Message Passing Library (MPL) als Basis moderner HPC-Anwendungen
- Vorteile der MPL gegenüber MPI:
  - erhöhte Typsiche
  - Compile-Time- st
  - automatisches Re
    (automatische Fre
  - automatisches Ba
  - praktisch kein Ov ionen und
    Template-Metapr
  - teils erhebliche R
- weiterhin bestehen
  - Empfänger muss er bereitstellen
  - mangelnde Möglichkeiten zu Reflexion/Introspektion in C++
  - Datentypen mit Elementen dynamischer Größe schlecht unterstützt
  - kaum Toleranz gegenüber Run-Time-Fehler
  - 32-Bit-Integer-API-Desaster

Fork me on GitHub