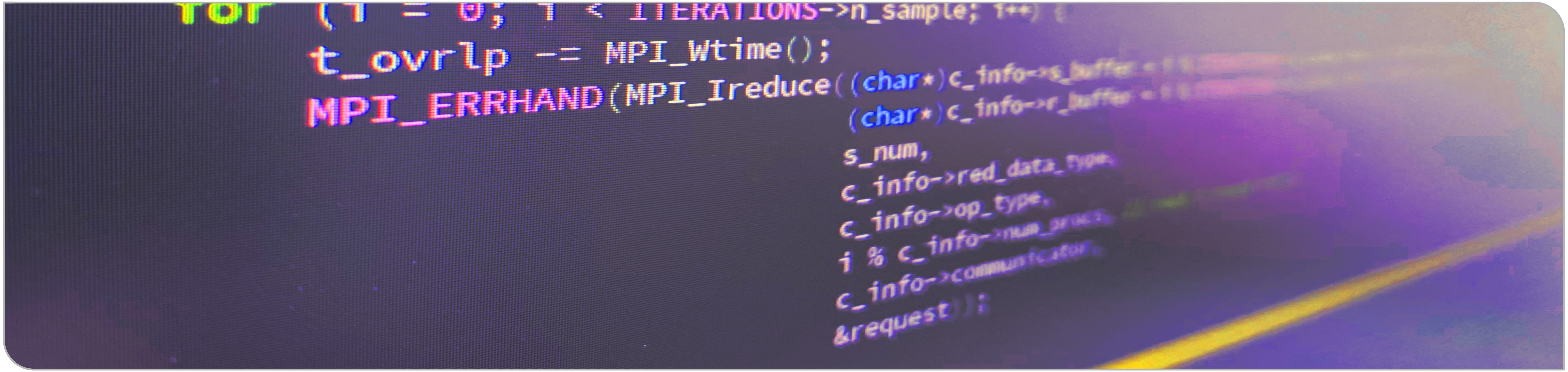
KIT
Karlsruhe Institute of Technology

(Near) Zero-Overhead C++ Bindings for MPI

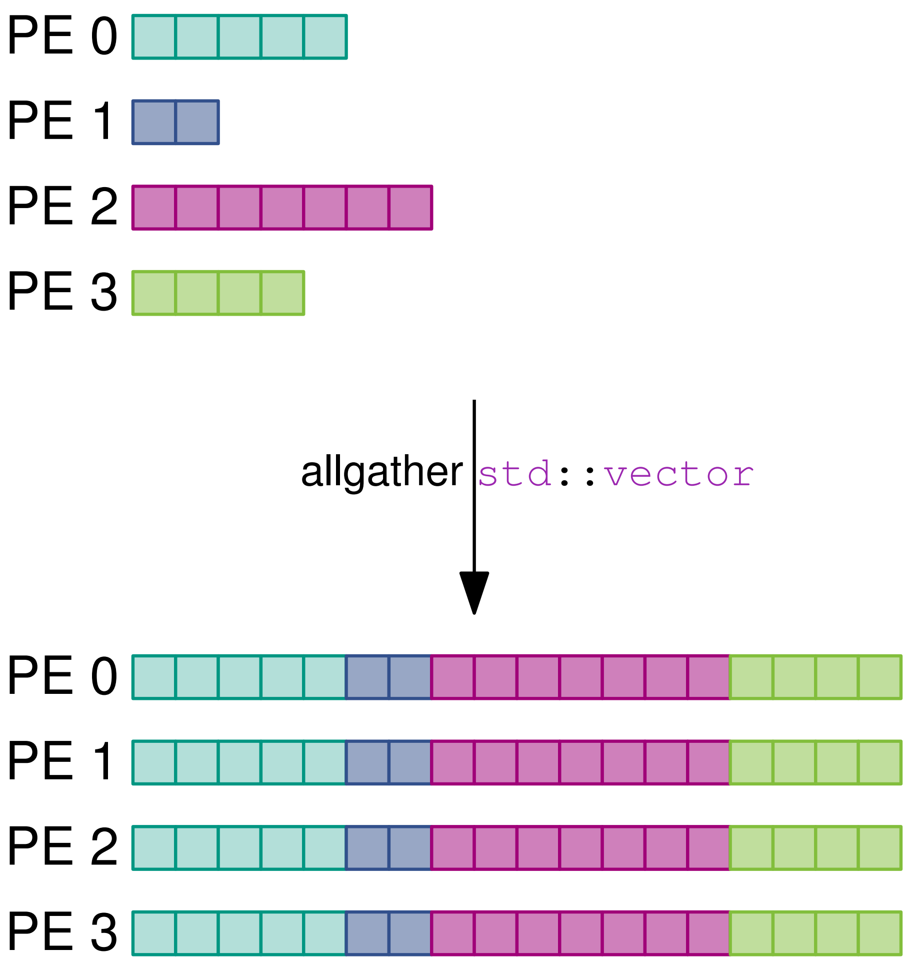
Brief Announcement @ SPAA'24 · June 19, 2024

Demian Hespe, Lukas Hübner, Florian Kurpicz, Peter Sanders,
Matthias Schimek, Daniel Seemaier, **Tim Niklas Uhl**

KaMPIng

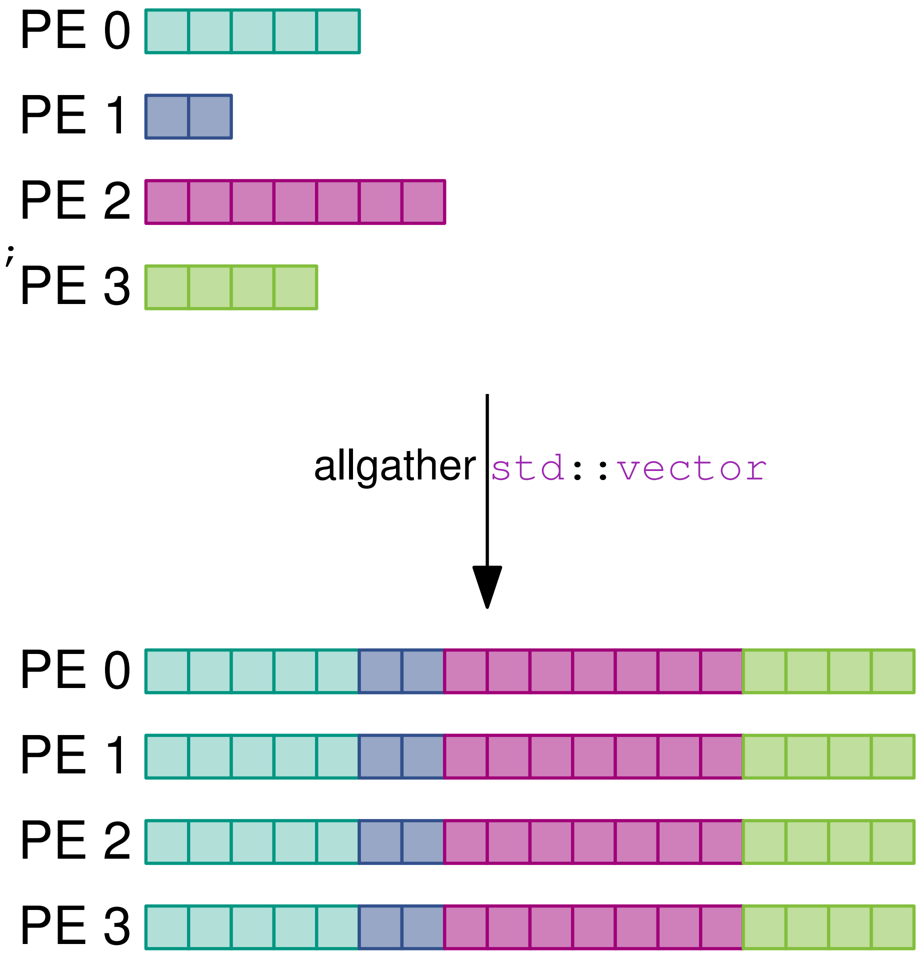


Using MPI from C++



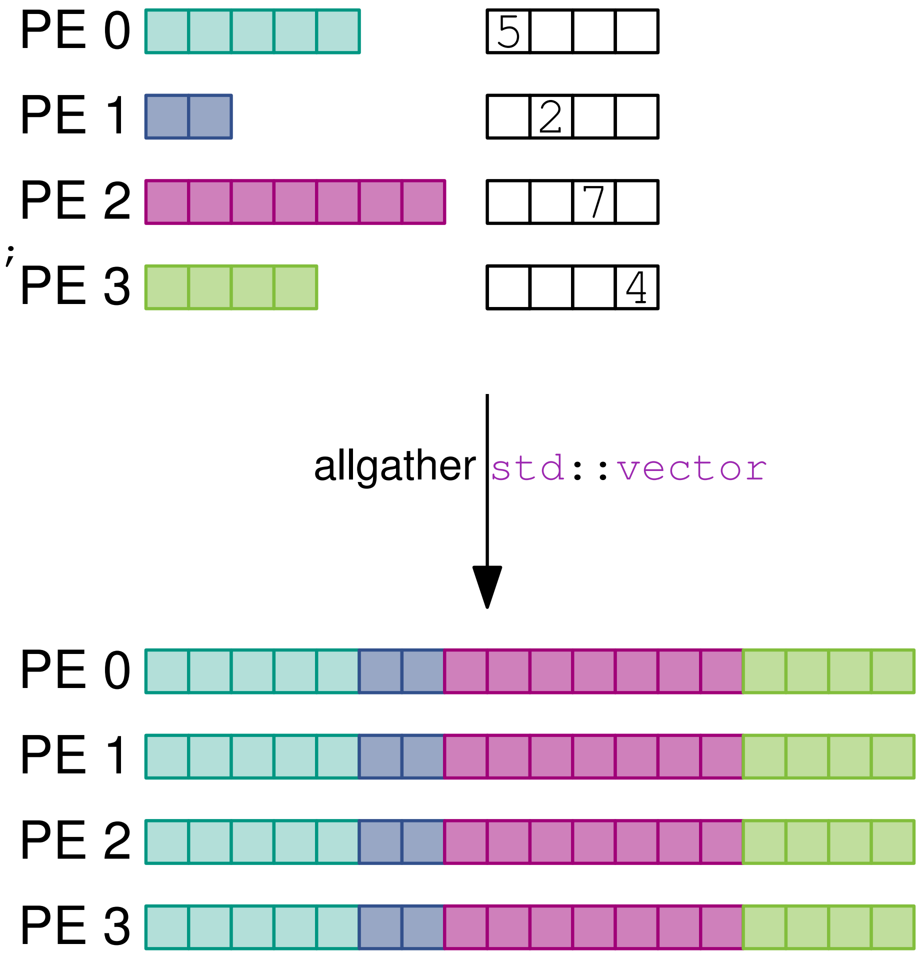
Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



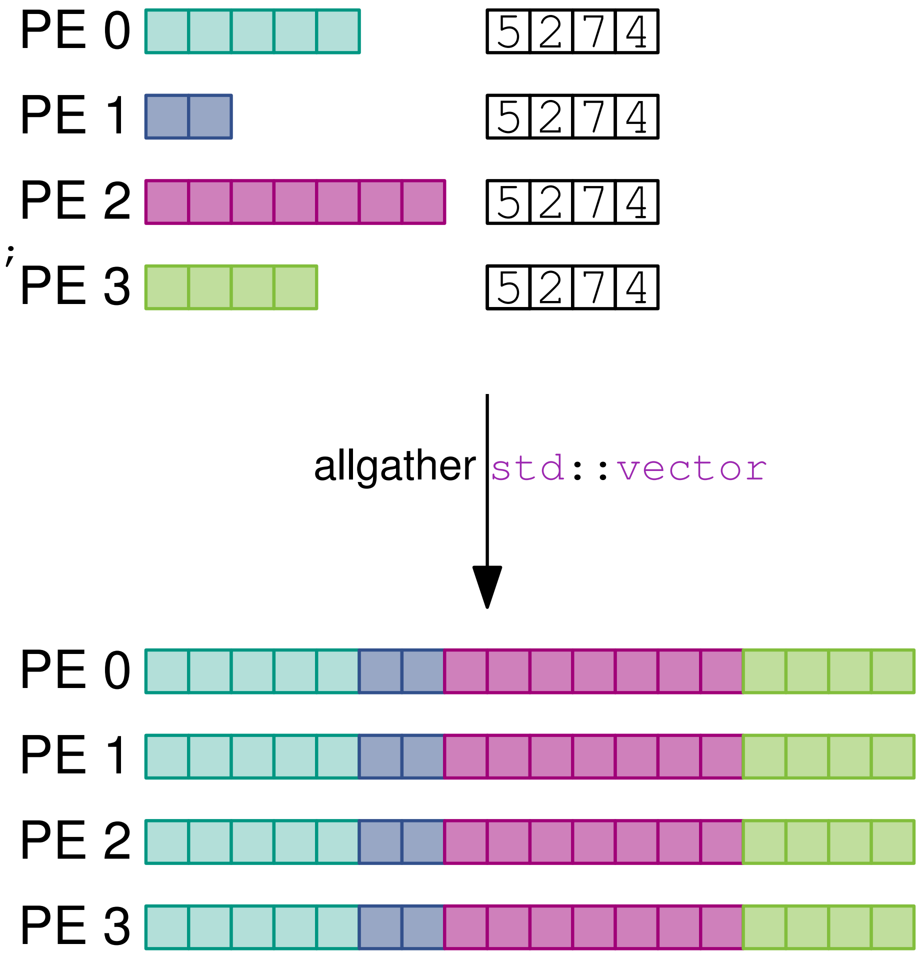
Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



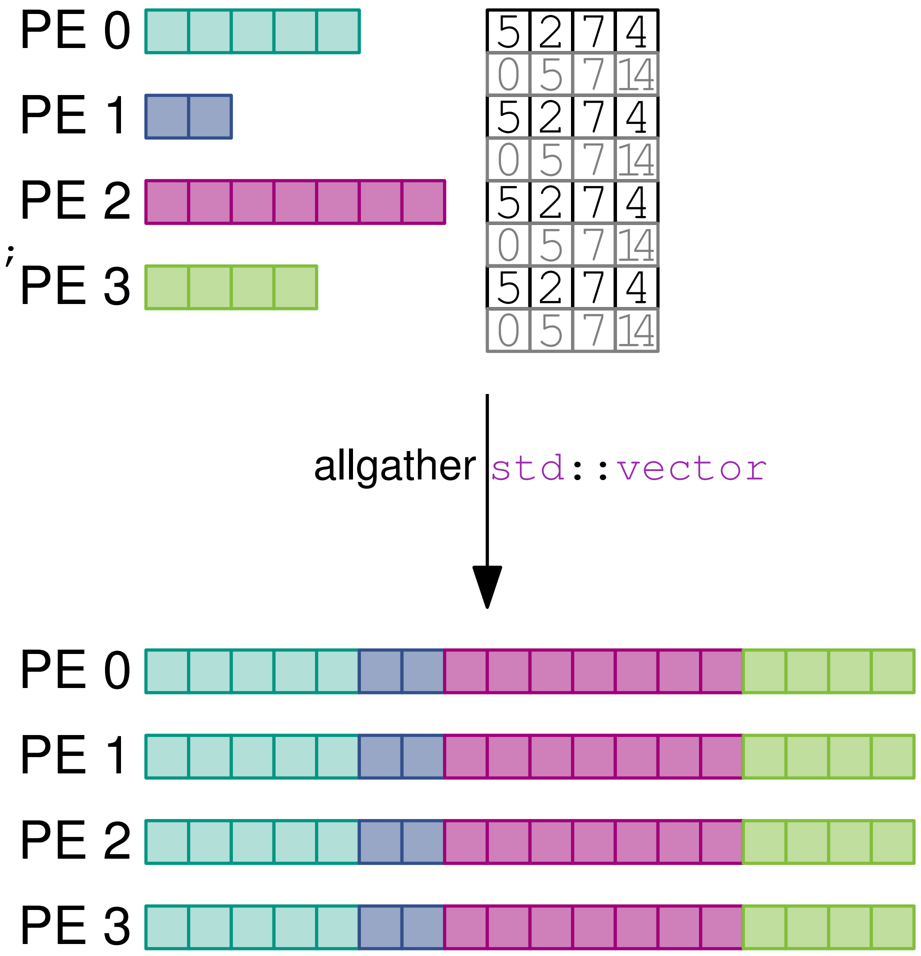
Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



Using MPI from C++

Karlsruhe Institute of Technology

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

C-ish API

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++


```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global(rd.back() + rc.back());
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

all other parameters can be inferred

parameter order?

arbitrary parameter order!

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global(rd.back() + rc.back());
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

manual allocation

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {


  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

common idiom: boilerplate!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe **MPI** next **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {


  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

common idiom: boilerplate!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {

  std::vector<int> rc(comm.size());
  rc[rank] = v_local.size();
  comm.allgather(send_recv_buf(rc));

  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc));

  return v_global;
}
```

common idiom: boilerplate!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe **MPI** next **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping $\leftrightarrow$ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global));


  return v_global;
}
```

return by reference

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  return comm.allgatherv(send_buf(v_local));


}
```

return by reference
or by value

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

Karlsruhe Institute of Technology

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {
  return comm.allgatherv(send_buf(v_local));
}
```

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {
  return comm.allgatherv(send_buf(v_local));
}
```

```
// avoid implicit allocation
comm.allgatherv(send_buf(v_local),
                recv_counts_out<no_resize>(some_buf));

// pass buffer ownership to calls
rc = comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                     recv_counts_out<resize_to_fit>(std::move(rc)));

// retrieve auxiliary data
auto [recvbuf, counts] = comm.allgatherv(send_buf(v_local),
                                         recv_counts_out());
```

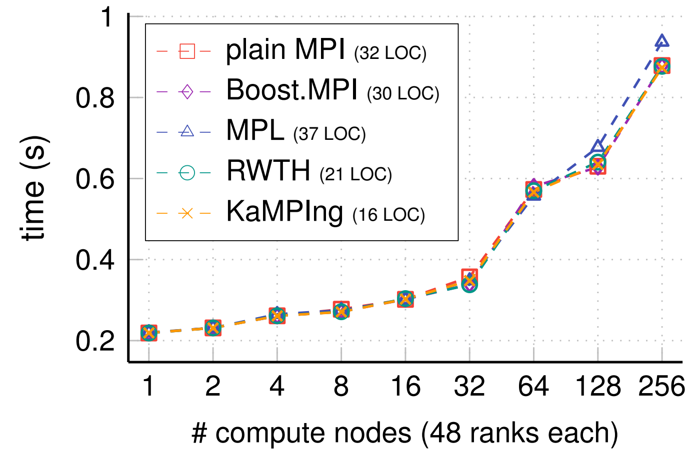
Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Conclusion

- **low-to-high-level** C++ bindings for MPI
- no runtime-overhead
- reduce boilerplate and error-proneness in MPI applications
  - default parameters
  - safety guarantess
  - fine-grained memory management
- base for a future **standard library** of distributed algorithms and data structures



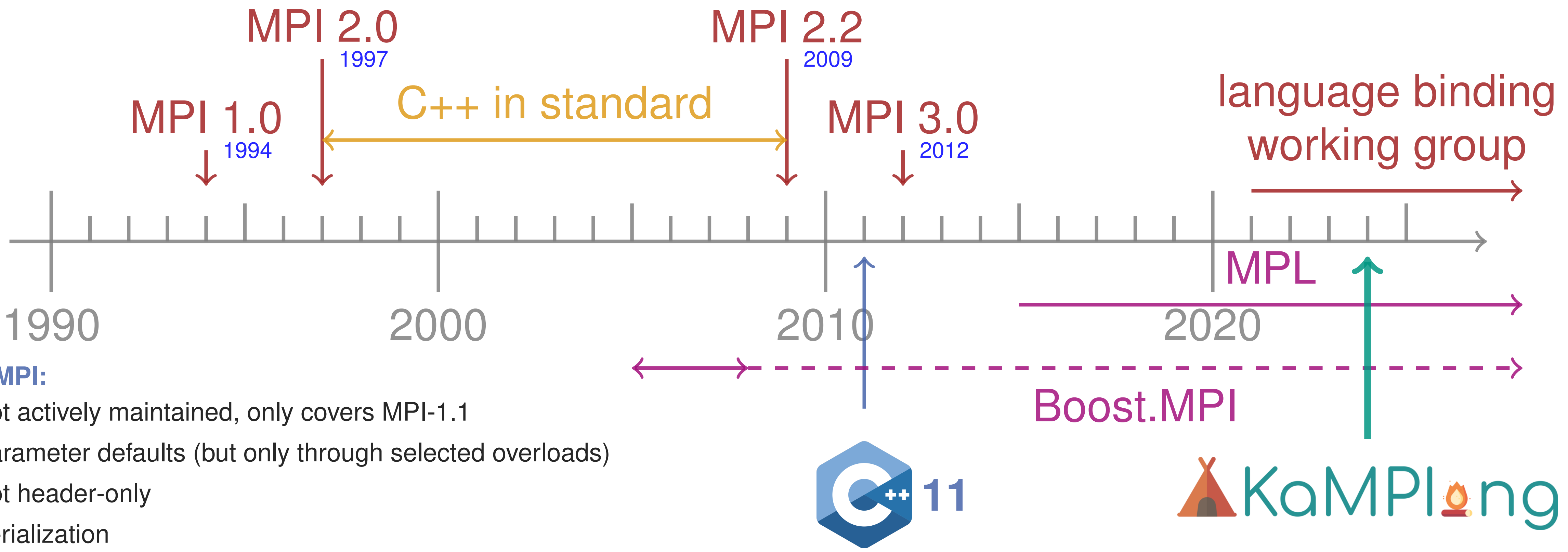
application benchmarks:
- phylogenetic interference
- graph analysis/partitioning
- (string) sorting

github.com/kamping-site/kamping

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

History of MPI and C++

Karlsruhe Institute of Technology



Boost.MPI:

– not actively maintained, only covers MPI-1.1
+ parameter defaults (but only through selected overloads)
– not header-only
+ serialization
– type system tightly coupled with serialization $\longrightarrow$ performance pitfalls

MPL:

+ powerful type system, suitable for scientific computing
– more complex irregular collectives via custom types $\longrightarrow$ less performant
– low abstraction level

More Features

Serialization

```
using dict = std::unordered_map<std::string, std::string>;
dict data = ...;
comm.send(send_buf( as_serialized(data)));

dict recv_dict = comm.recv(
  send_buf( as_deserializable<dict>())
);
```

Memory Safety for Nonblocking Operations

```
std::vector<int> v = ...;
auto r1 = comm.isend(
    send_buf_out(std::move(v)), destination(1)
);

v = r1.wait(); // v is moved back to caller after
               // request is complete

auto r2 = comm.irecv<int>(recv_count(42));
std::vector<int> data = r2.wait(); // data only returned
                                   // after request
                                   // is complete
```

Specialized Collectives