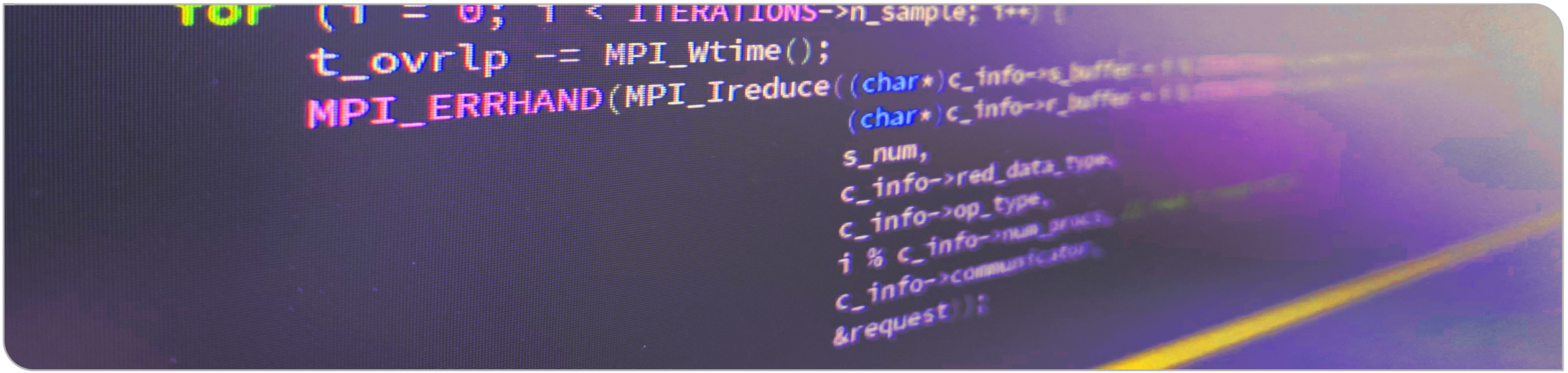
Karlsruhe Institute of Technology

Scalable Discrete Algorithms for the Basic Toolbox

HLRS Results and Review Workshop · 2024-10-10

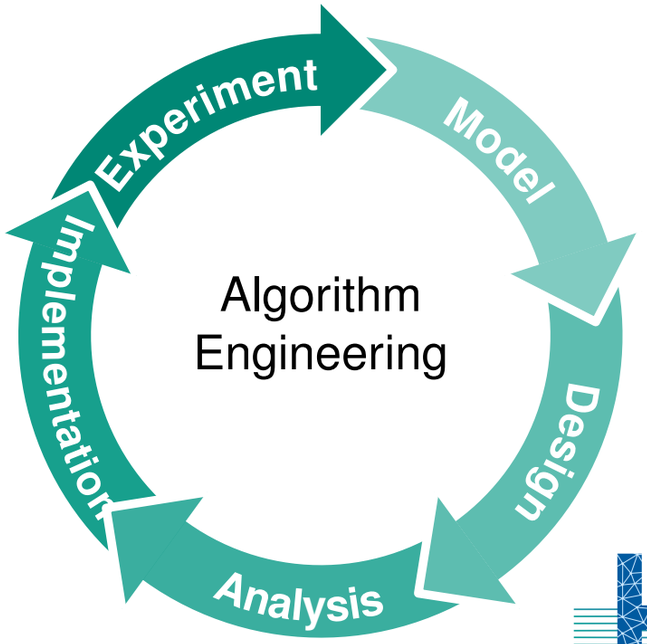
Lukas Hübner, Florian Kurpicz, Peter Sanders, Matthias Schimek,
Dominik Schreiber, Daniel Seemaier, **Tim Niklas Uhl**

Institute of Theoretical Informatics (ITI), Algorithm Engineering



Project Overview

Design **scalable algorithms** for essential building blocks in **massively parallel computing**, with a focus on non-numerical algorithms for **Big Data** applications



HoreKa

Basic Toolbox

Automated SATisfiability
- SAT Solving
- malleable job scheduling

Graph Algorithms
- Minimum Spanning Tree
- Connected Components
- Triangle Counting
- Graph Partitioning

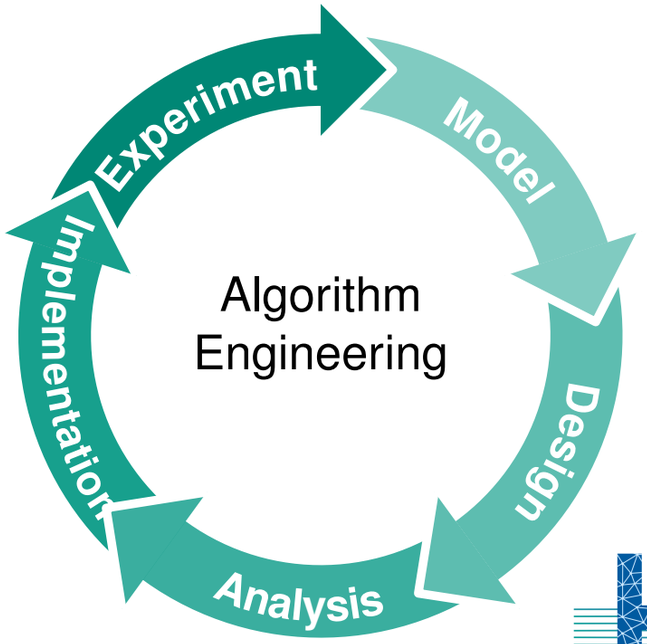
Text Indexing
- String Sorting

Fault Tolerance

Communication Primitives
- MPI C++ bindings

Project Overview

Design **scalable algorithms** for essential building blocks in **massively parallel computing**, with a focus on non-numerical algorithms for **Big Data** applications



HoreKa

Basic Toolbox

Automated SATisfiability
- SAT Solving
- malleable job scheduling

Graph Algorithms
- Minimum Spanning Tree
- Connected Components
- Triangle Counting
- Graph Partitioning

Text Indexing
- String Sorting

Fault Tolerance

Communication Primitives
- MPI C++ bindings

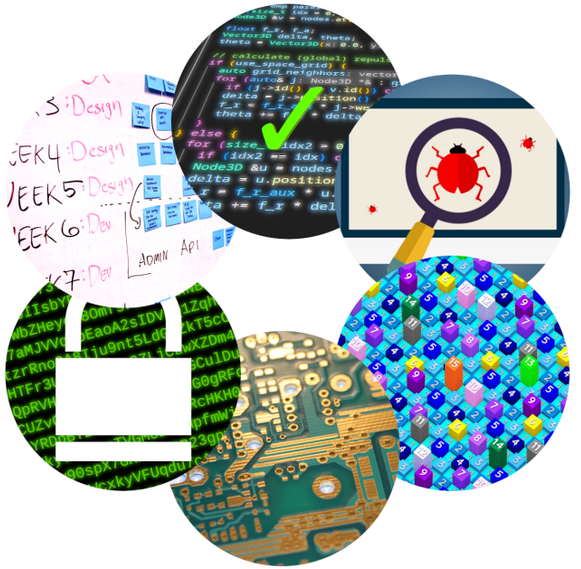
SAT Solving and Malleable Job Scheduling

Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$



SAT Solving and Malleable Job Scheduling

Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$

SAT Solving and Malleable Job Scheduling

Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

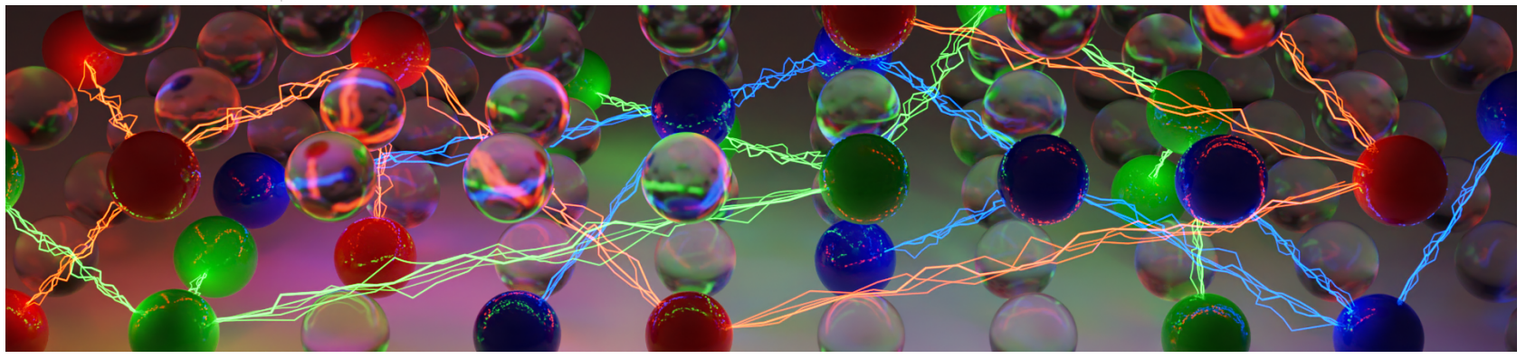
The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$

The mAllob system

schedule · balance · solve · prove



SAT Solving and Malleable Job Scheduling

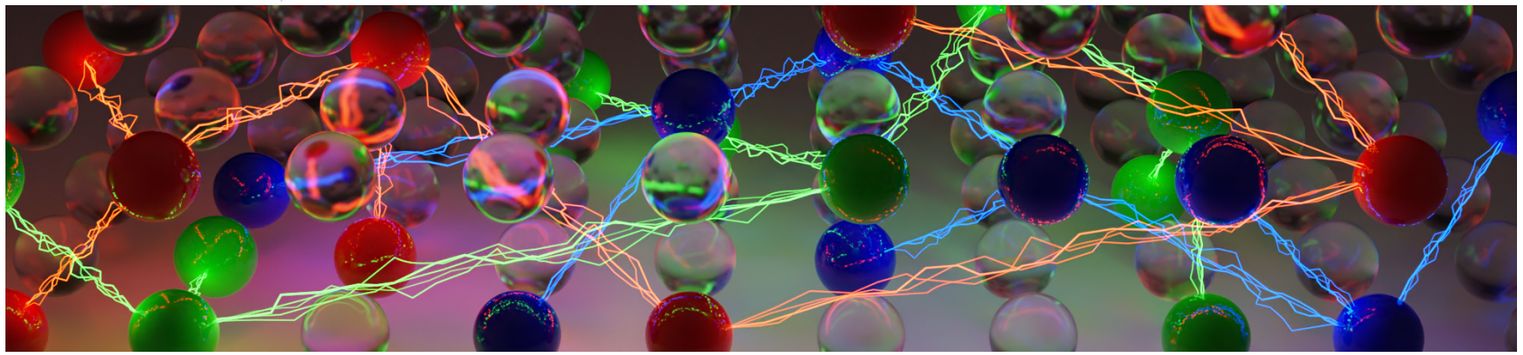
Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

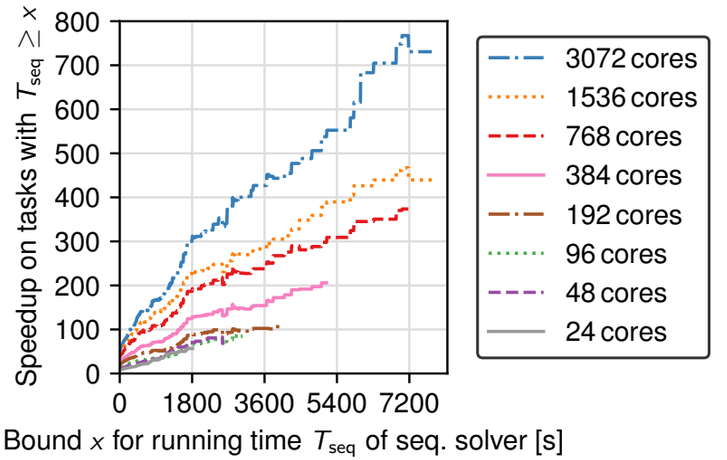
The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$

The mALLOB system





400 SAT instances

SAT Solving and Malleable Job Scheduling

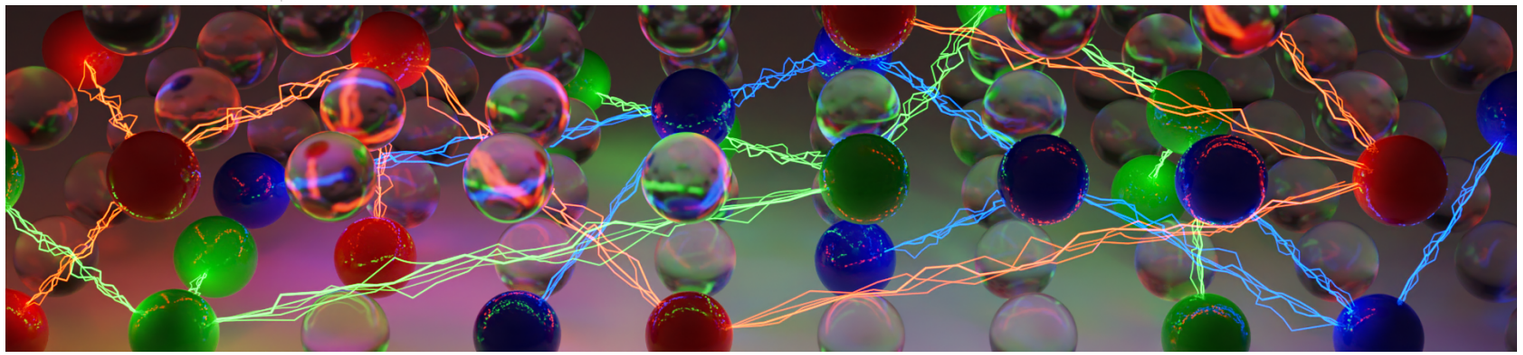
Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

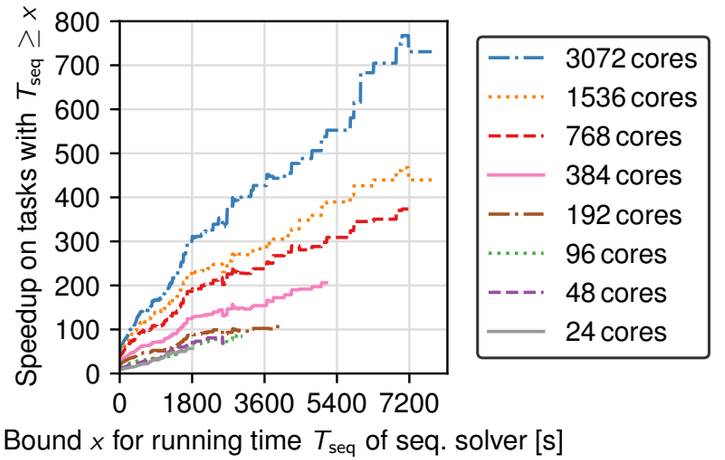
The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$

The mALLOB system





400 SAT instances

- Latest research: **Produce and validate proofs** that a result is correct
  $\Rightarrow$ Enables use of distributed SAT solving for **critical applications**, e.g., **software verification**

SAT Solving and Malleable Job Scheduling

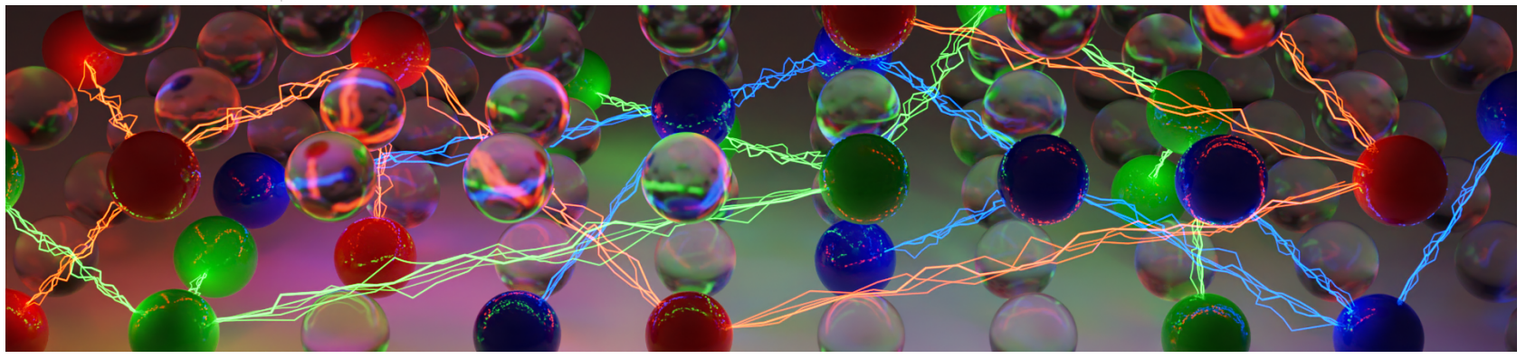
Sanders, Schreiber [SAT'20–24, Euro-Par'23, JAIR, SPAA'24, HOPC'24]

The $\mathcal{NP}$-complete SAT problem [Cook 1971]

Find a **satisfying variable assignment** for **propositional formula**

$F := \bigwedge_{c \in C} (\bigvee_{\ell \in c} \ell)$

The mallob system



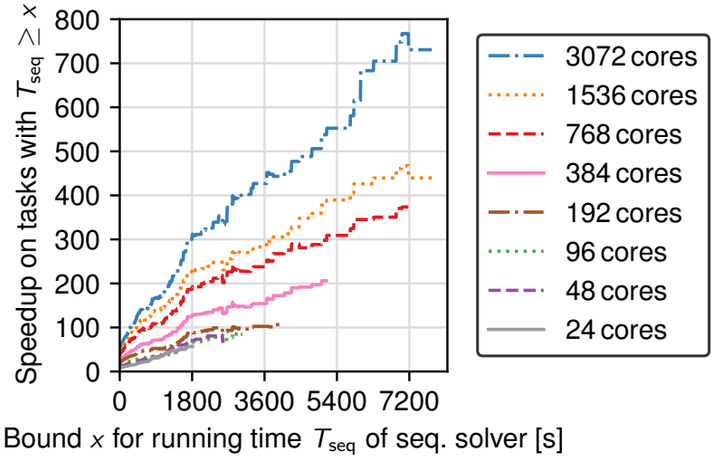
- Latest research: **Produce and validate proofs** that a result is correct
  ⇒ Enables use of distributed SAT solving for **critical applications**, e.g., **software verification**

*"Mallob-mono is now, by a **wide** margin, the most powerful SAT solver on the planet."* —Byron Cook, Amazon Science, 2021

Best cloud solver @ SAT'20–24

Dissertation Award **Fahiem Bacchus Award**

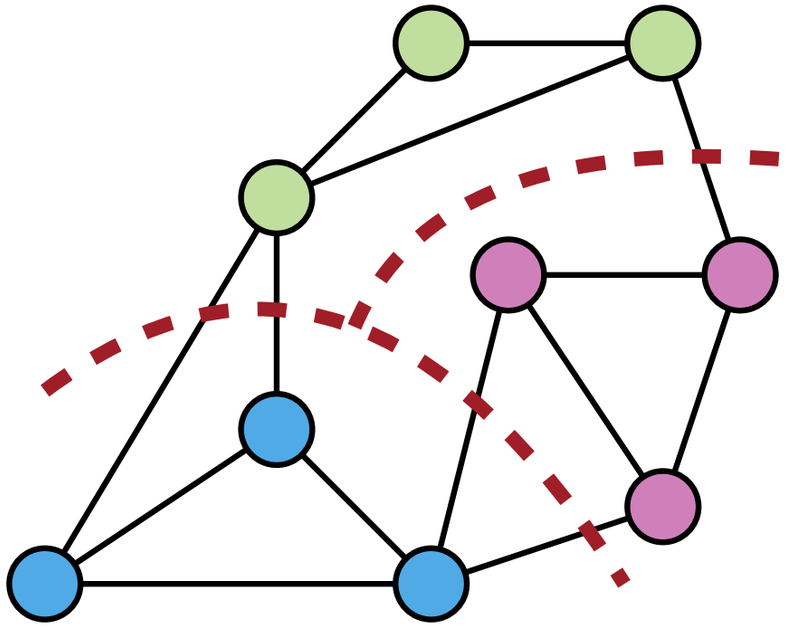
github.com/domschrei/mallob



400 SAT instances

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

- blocks have roughly the same weight:
  $c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$
- while minimizing some objective, e.g., edge cut:
  $\sum_{i \neq j} \omega(E_{ij})$

Nearest Neighbor Search

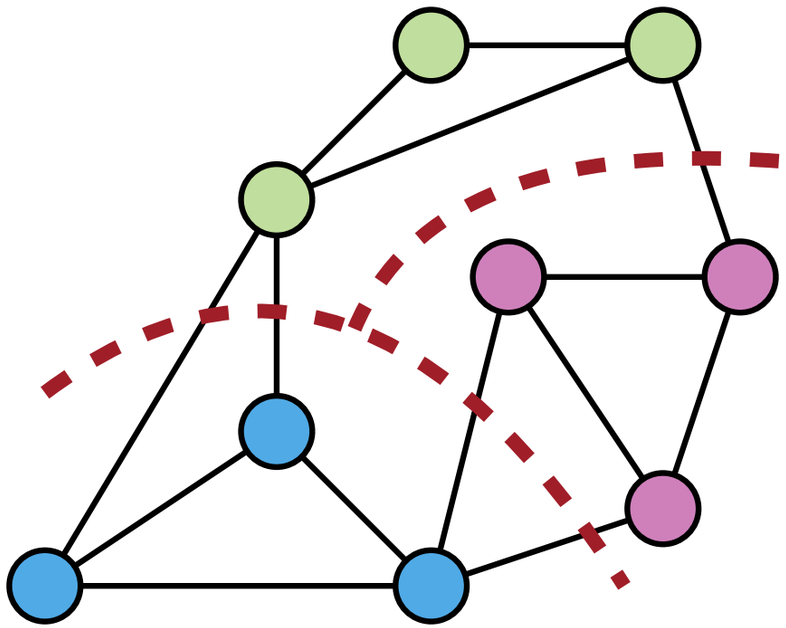
Distributed Databases

Circuit Placement

Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:
  $$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$
- while minimizing some objective, e.g., edge cut:
  $$\sum_{i \neq j} \omega(E_{ij})$$

Nearest Neighbor Search

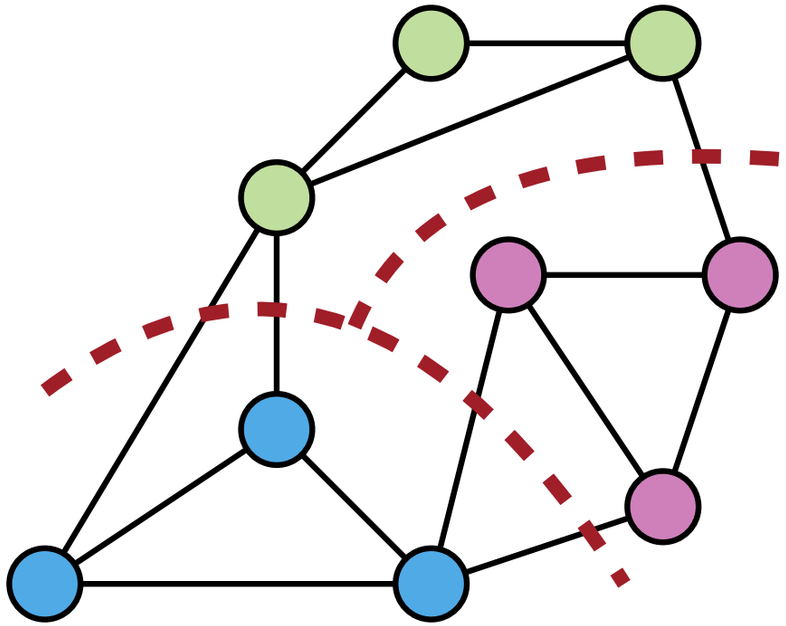
Distributed Databases

Circuit Placement

Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:

$$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$

imbalance factor

- while minimizing some objective, e.g., edge cut: $\sum_{i \neq j} \omega(E_{ij})$

Nearest Neighbor Search

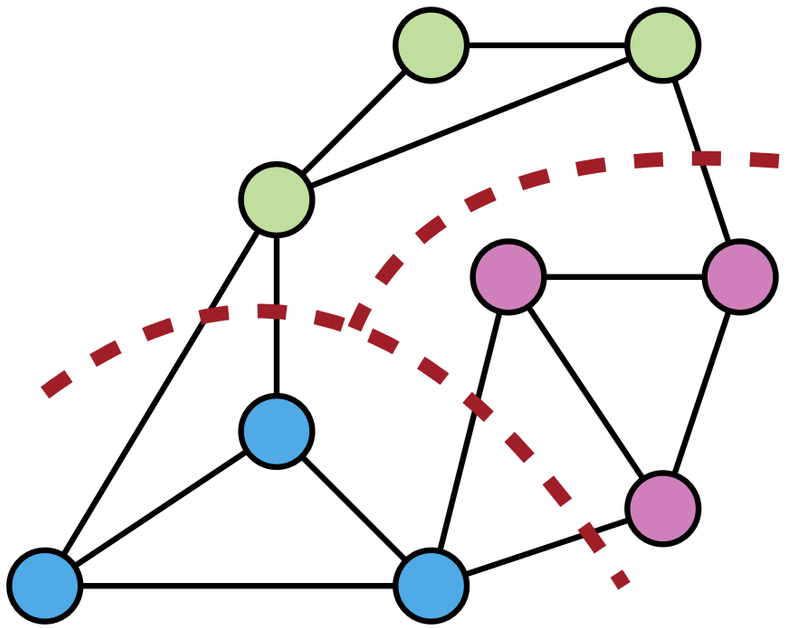
Distributed Databases

Circuit Placement

Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:

$$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$

imbalance factor

- while minimizing some objective, e.g., edge cut:

$$\sum_{i \neq j} \omega(E_{ij})$$

edges between blocks i and j

Nearest Neighbor Search

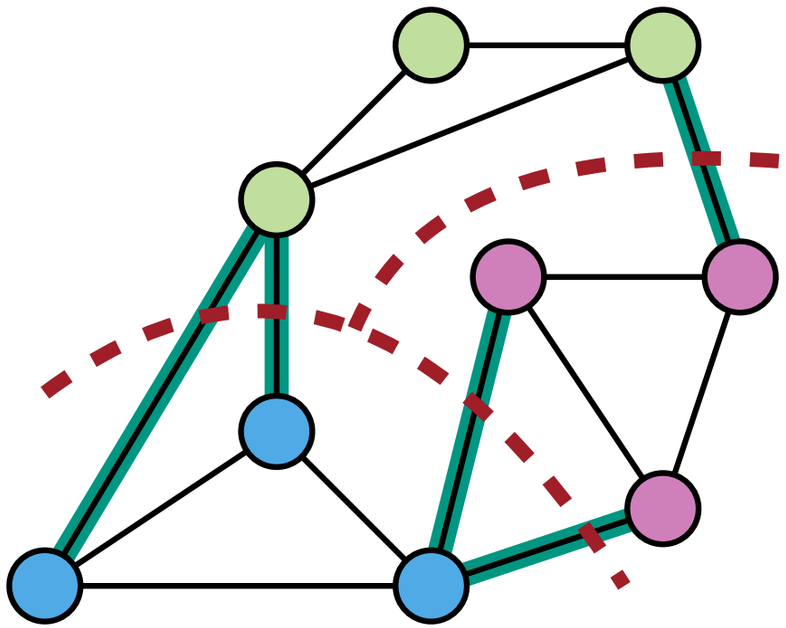
Distributed Databases

Circuit Placement

Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:

$$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$

imbalance factor

- while minimizing some objective, e.g., edge cut:

$$\sum_{i \neq j} \omega(E_{ij}) = 5$$

edges between blocks i and j

Nearest Neighbor Search

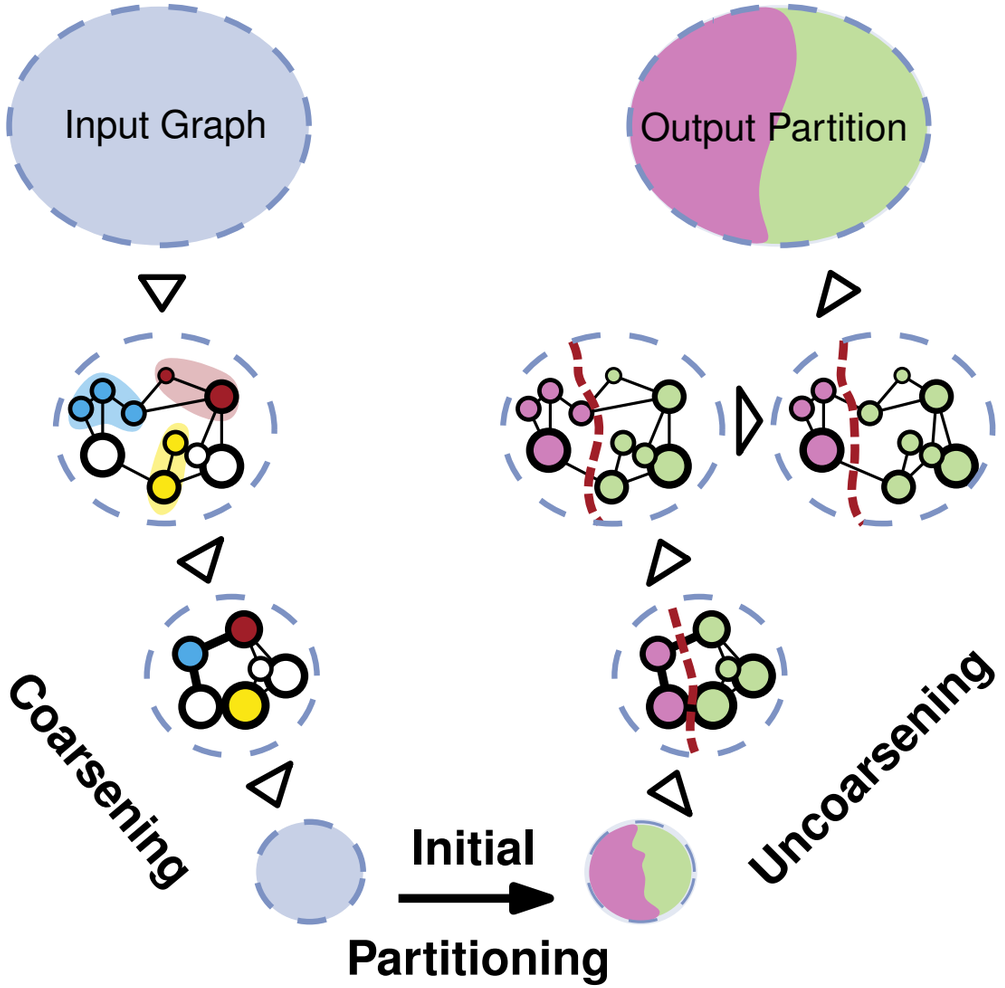
Distributed Databases

Circuit Placement

Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:

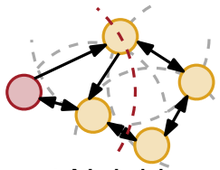
$$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$

imbalance factor

- while minimizing some objective, e.g., edge cut:

$$\sum_{i \neq j} \omega(E_{ij})$$

edges between blocks i and j


Nearest Neighbor Search

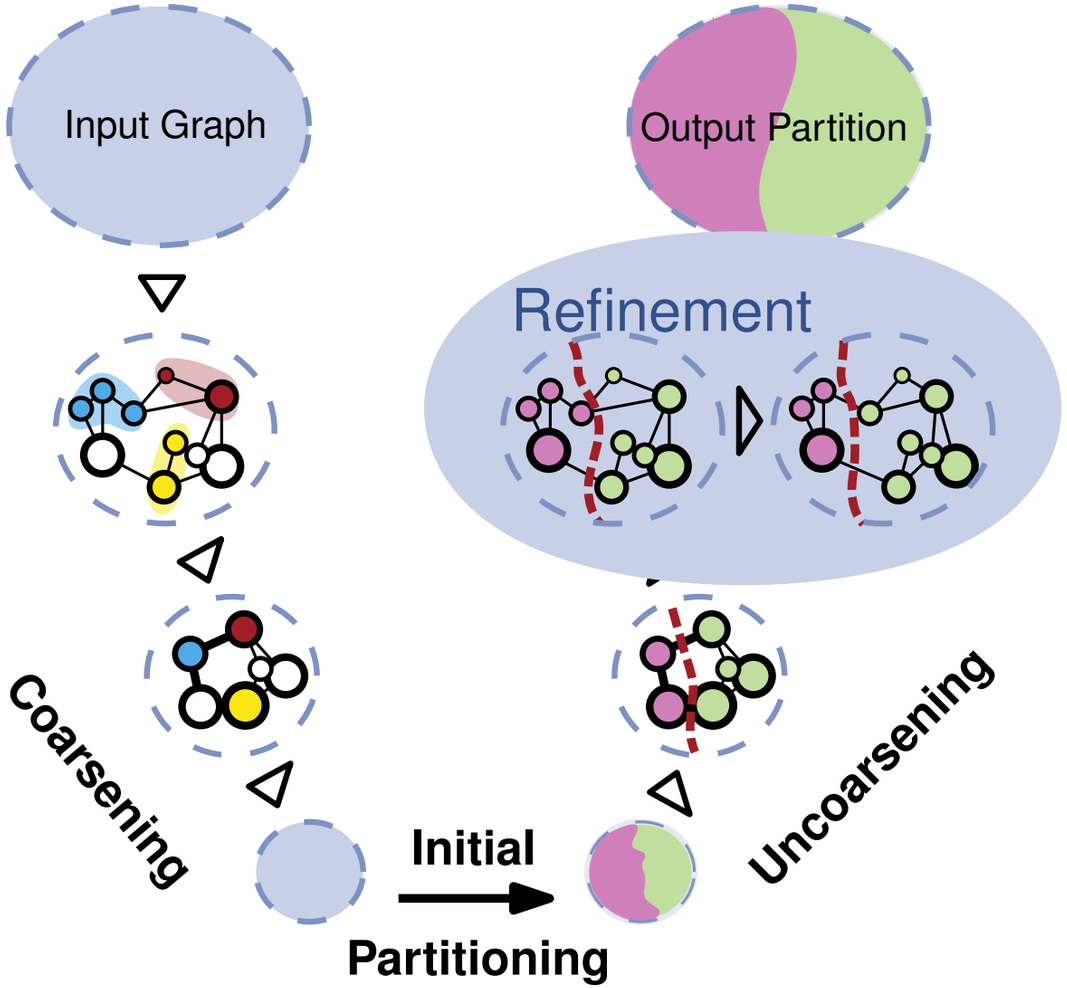

Distributed Databases


Circuit Placement


Load Balancing

Scalable Distributed Graph Partitioning

Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]



Given a graph $G = (V, E, c, \omega)$, partition V into k disjoint blocks such that:

vertex weights — edge weights

- blocks have roughly the same weight:

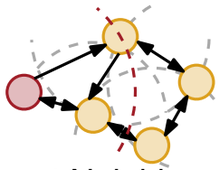
$$c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$$

imbalance factor

- while minimizing some objective, e.g., edge cut:

$$\sum_{i \neq j} \omega(E_{ij})$$

edges between blocks i and j


Nearest Neighbor Search


Distributed Databases


Circuit Placement

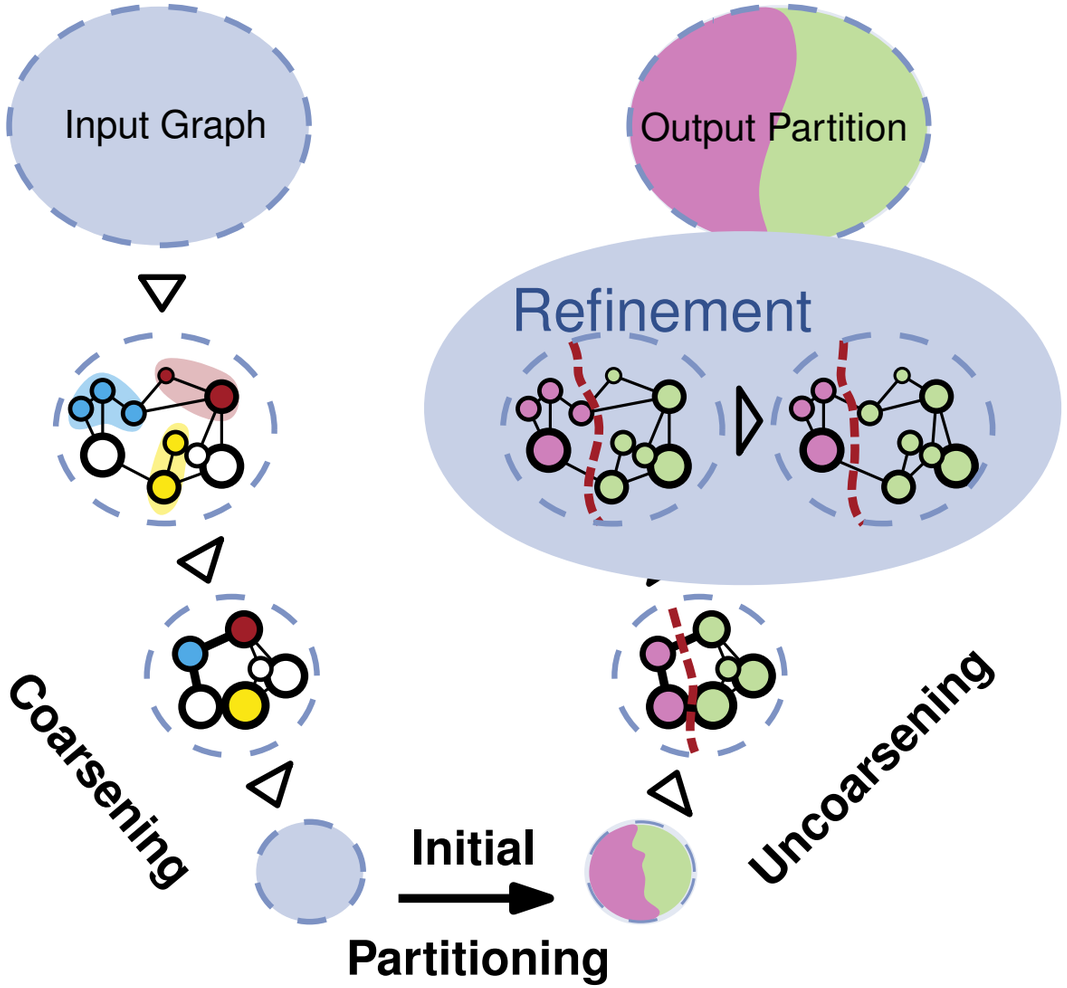

Load Balancing

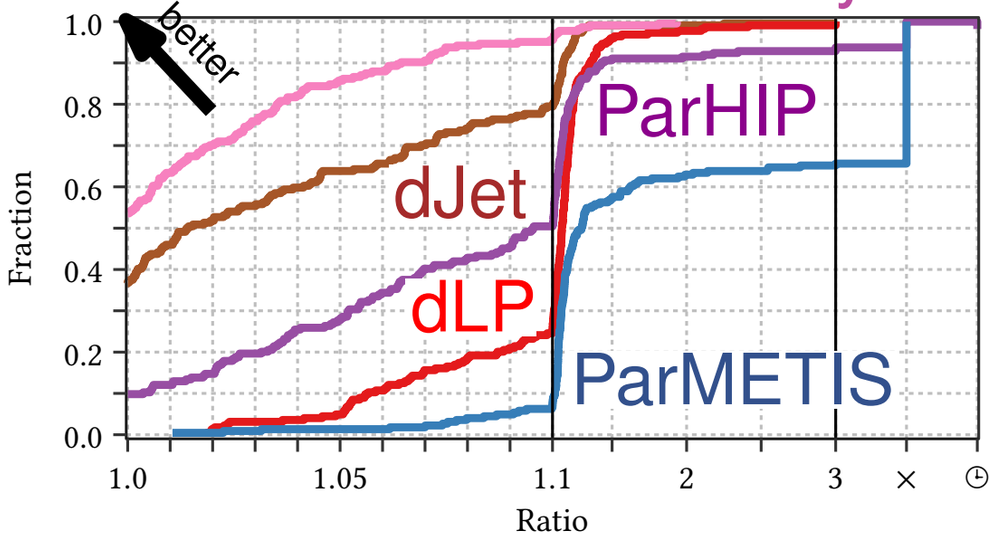
Scalable Distributed Graph Partitioning

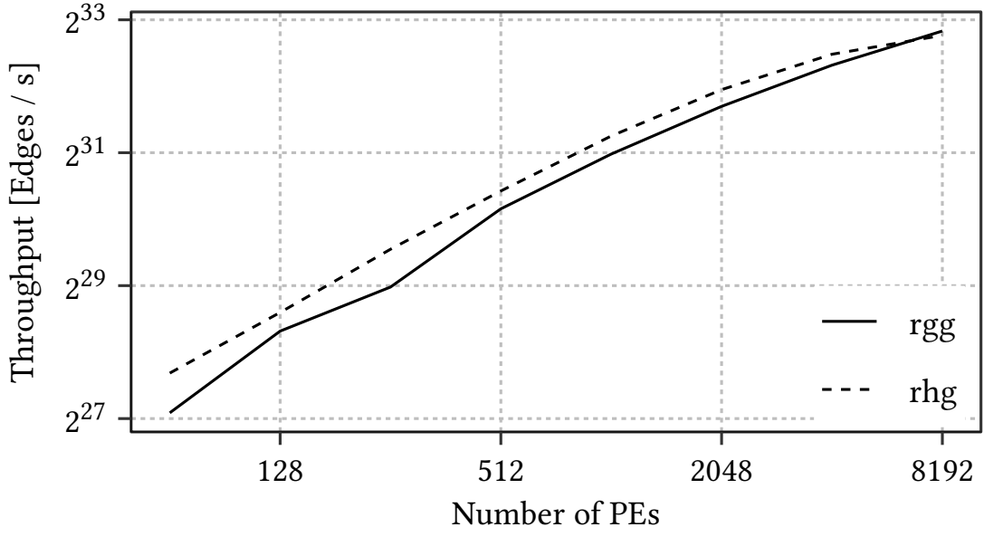
Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]

Karlsruhe Institute of Technology

Mt-KaHyPar





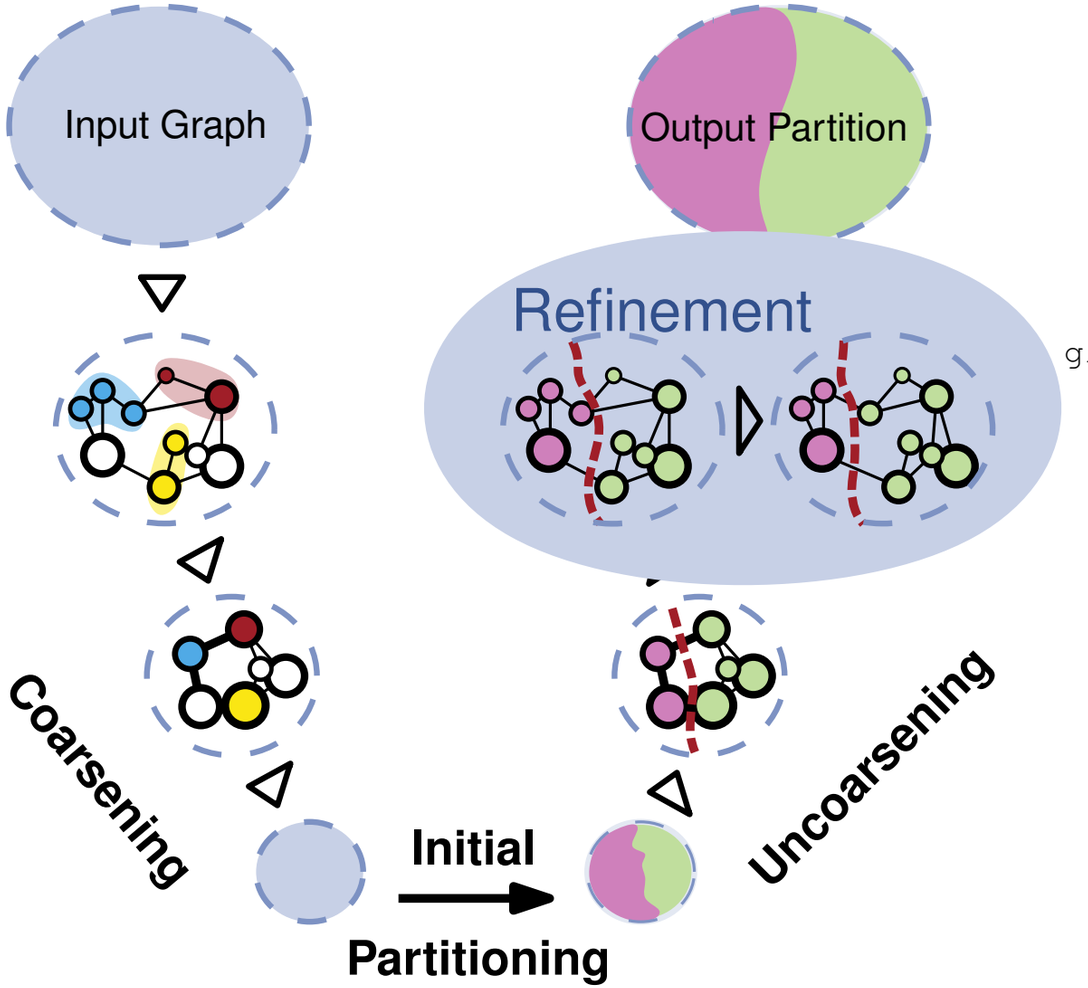


Scalable Distributed Graph Partitioning

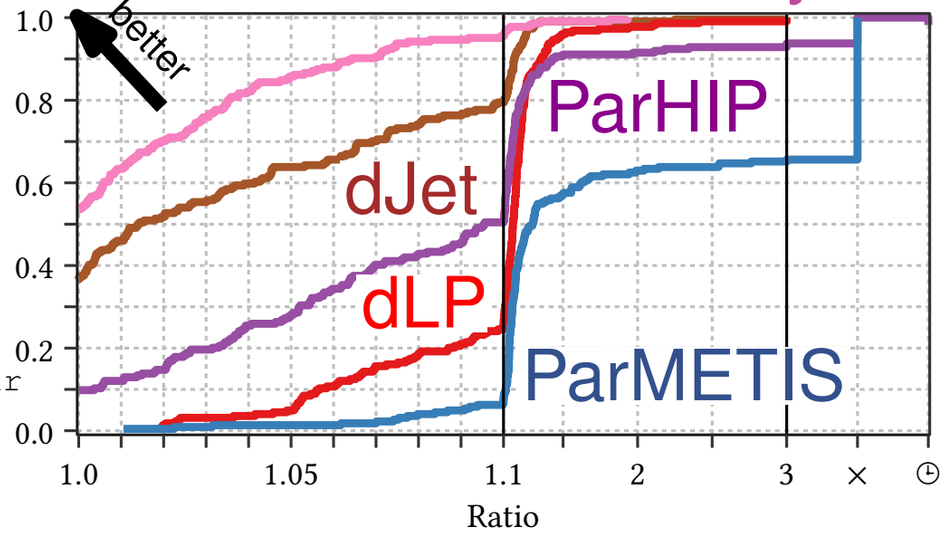
Sanders, Seemaier [Euro-Par'23, SPAA'24, HOPC'24]

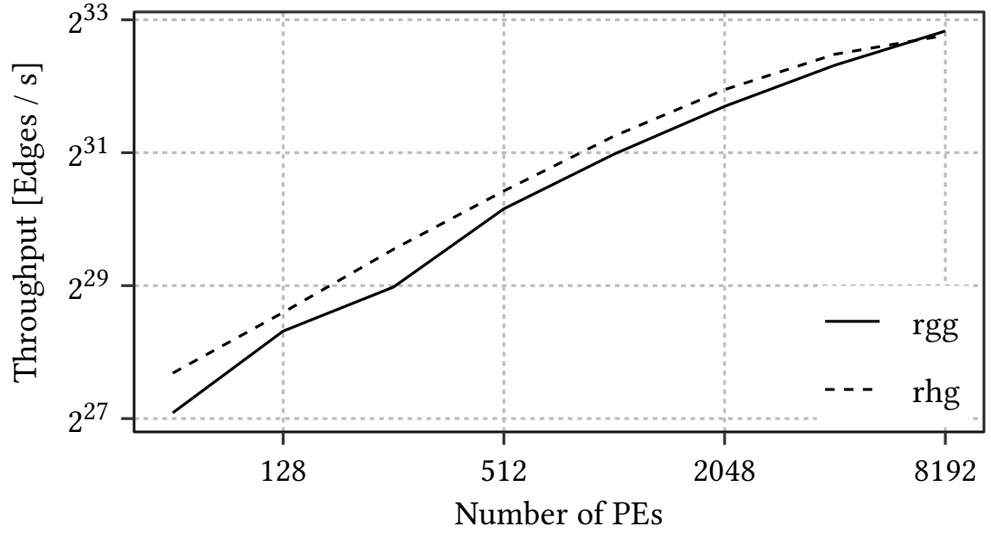
Karlsruhe Institute of Technology

Mt-KaHyPar



github.com/KaHIP/KaMinPar



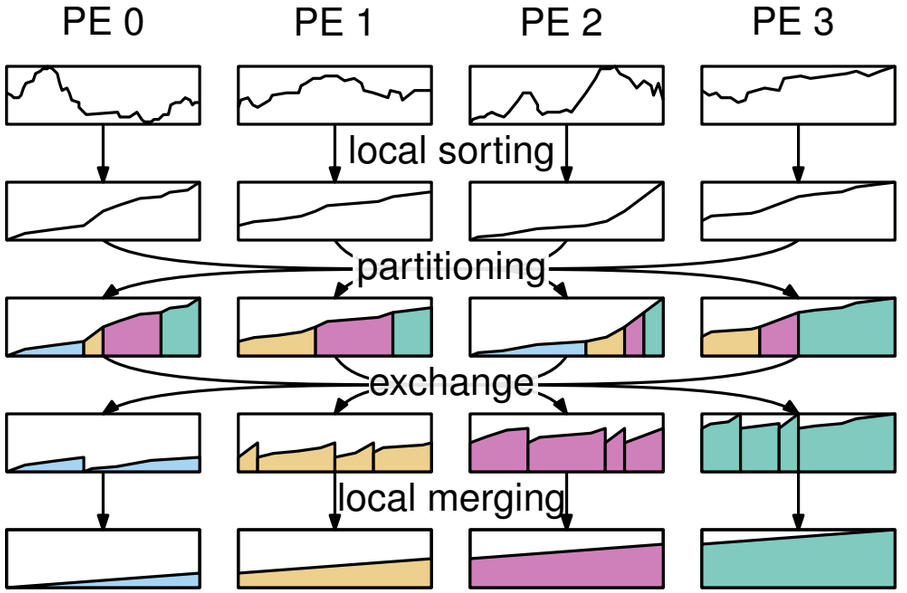


Scalable Distributed String Sorting

Kurpicz, Mehnert, Sanders, Schimek [SPAA'24, ESA'24]

- sorting strings is **multidimensional** problem
- different from classical **atomic** sorting
- algorithm: distributed merge string sort

$\Rightarrow$ low comm. volume **but high latency**



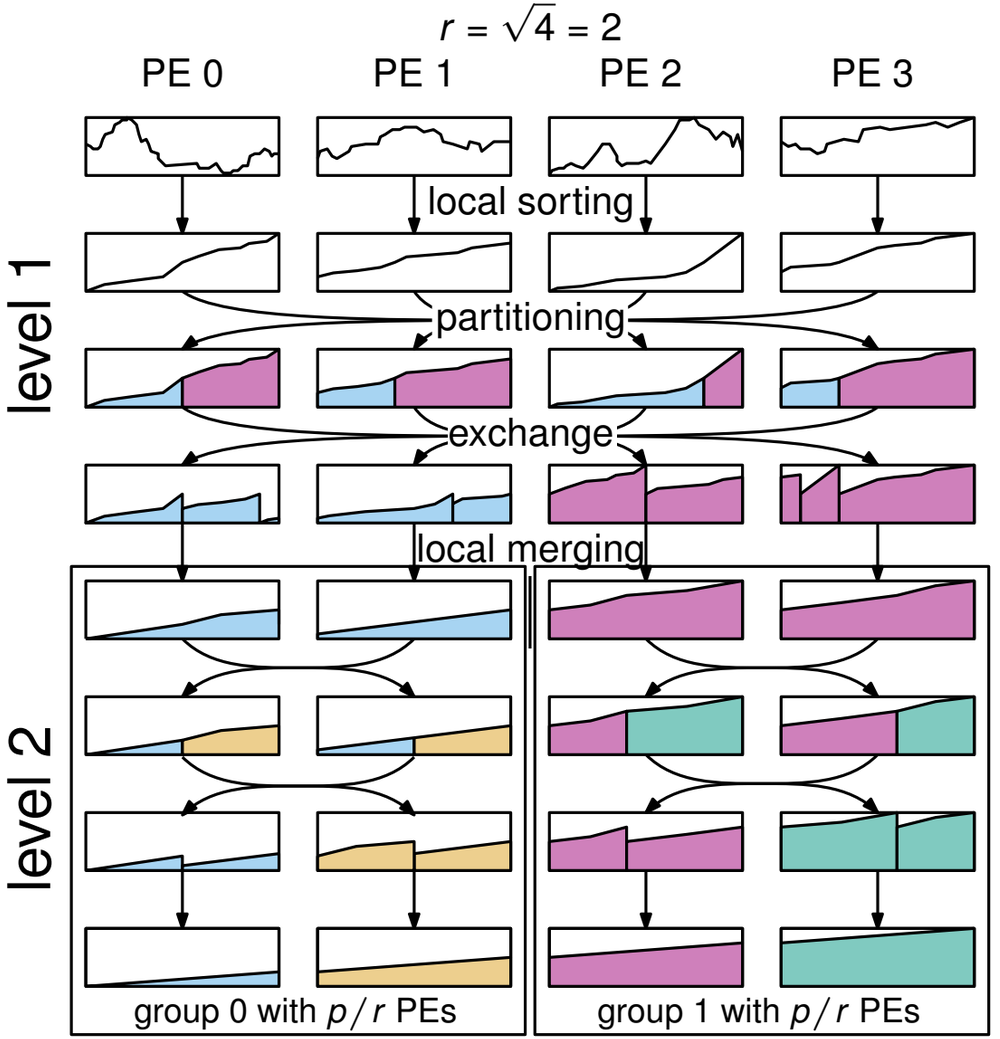
Scalable Distributed String Sorting

Kurpicz, Mehnert, Sanders, Schimek [SPAA'24, ESA'24]

- sorting strings is **multidimensional** problem
- different from classical **atomic** sorting
- algorithm: distributed merge string sort

$\Rightarrow$ low comm. volume **but high latency**

Solution: multiple recursion levels, only $\mathcal{O}(r)$ communication partners



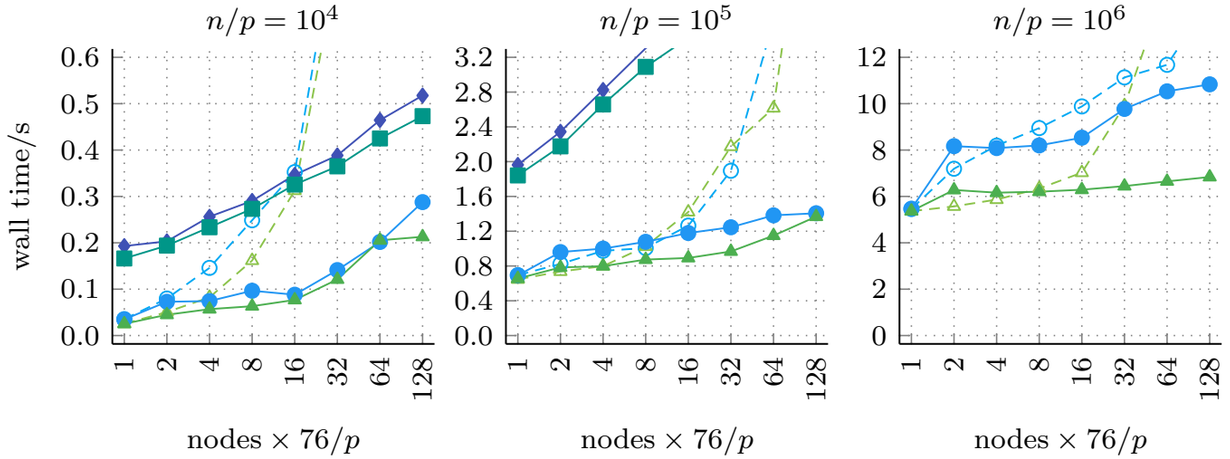
Scalable Distributed String Sorting

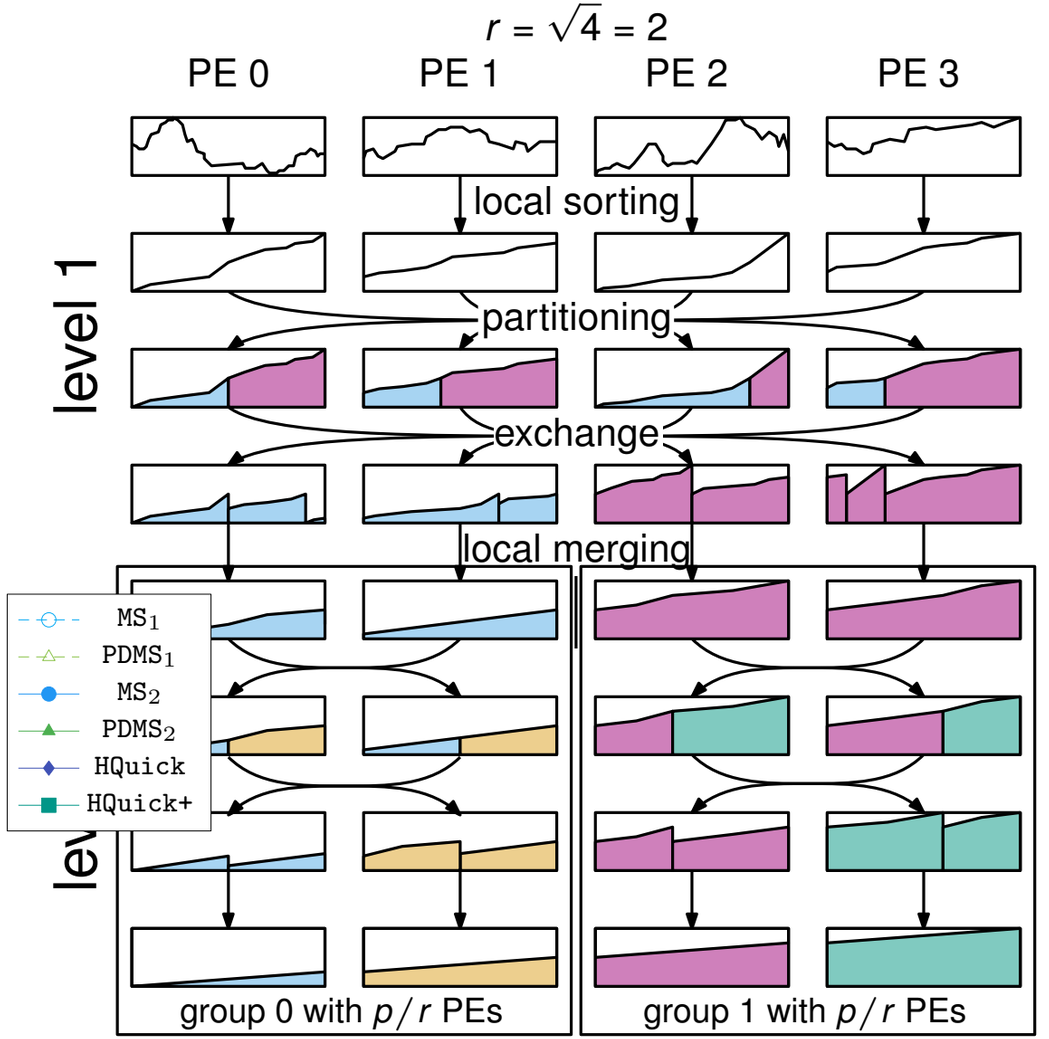
Kurpicz, Mehnert, Sanders, Schimek [SPAA'24, ESA'24]

- sorting strings is **multidimensional** problem
- different from classical **atomic** sorting
- algorithm: distributed merge string sort

$\Rightarrow$ low comm. volume **but high latency**

Solution: multiple recursion levels, only $\mathcal{O}(r)$ communication partners





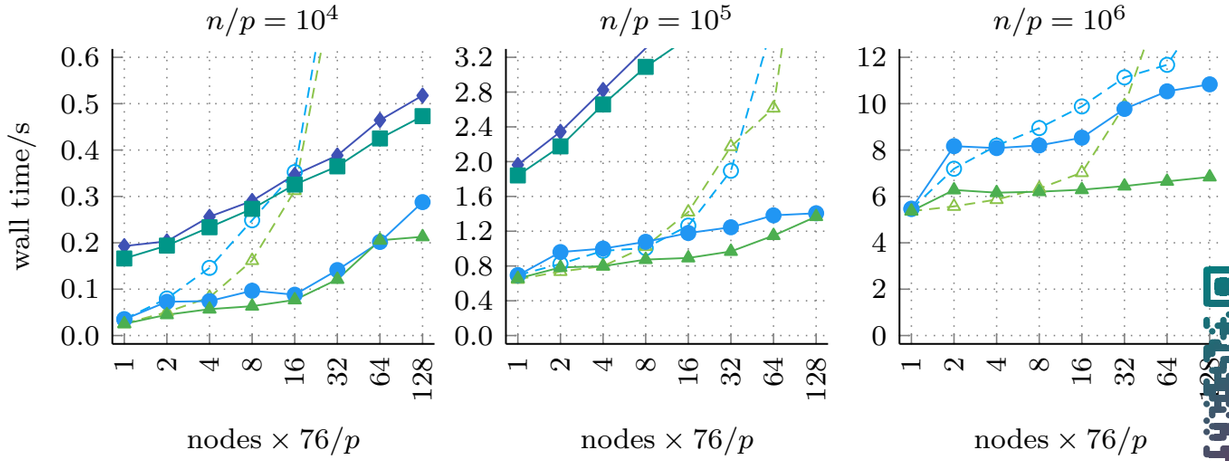
Scalable Distributed String Sorting

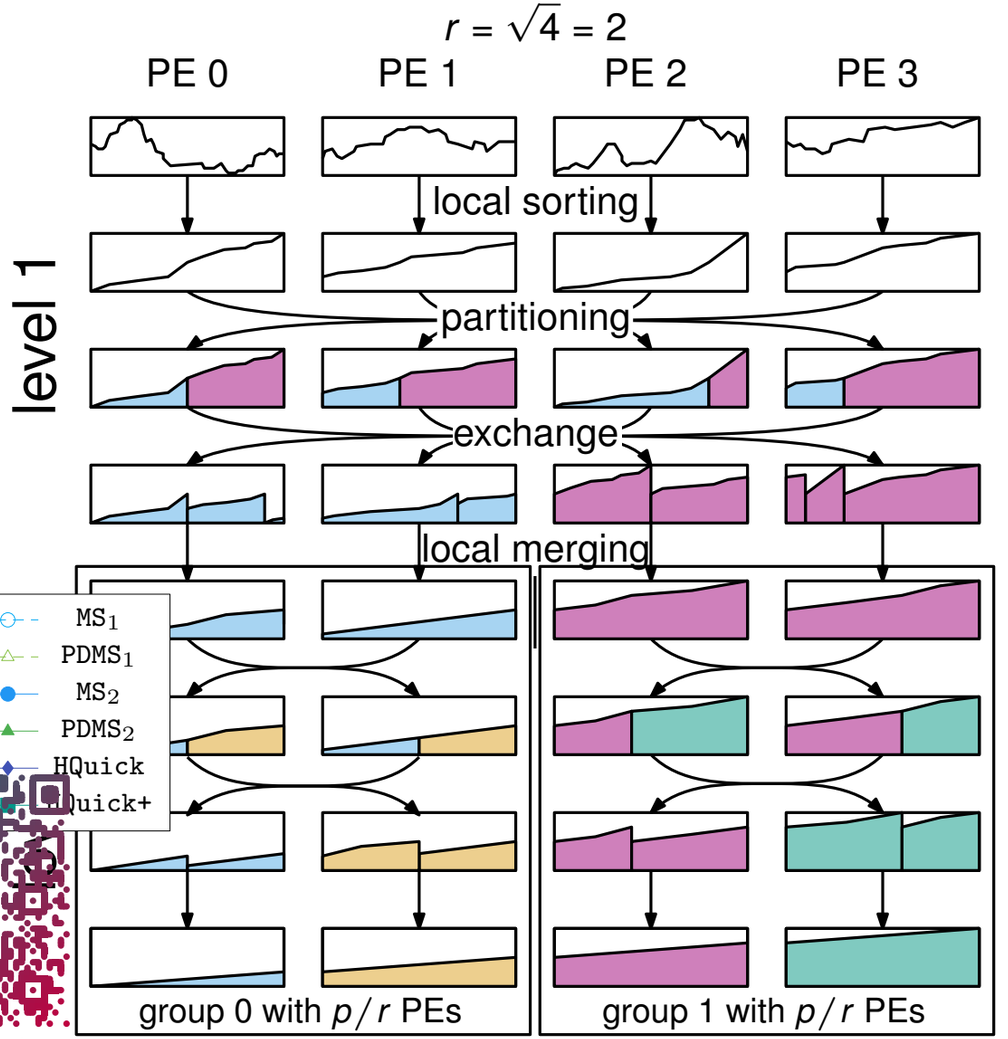
Kurpicz, Mehnert, Sanders, Schimek [SPAA'24, ESA'24]

- sorting strings is **multidimensional** problem
- different from classical **atomic** sorting
- algorithm: distributed merge string sort

$\Rightarrow$ low comm. volume **but high latency**

Solution: multiple recursion levels, only $\mathcal{O}(r)$ communication partners

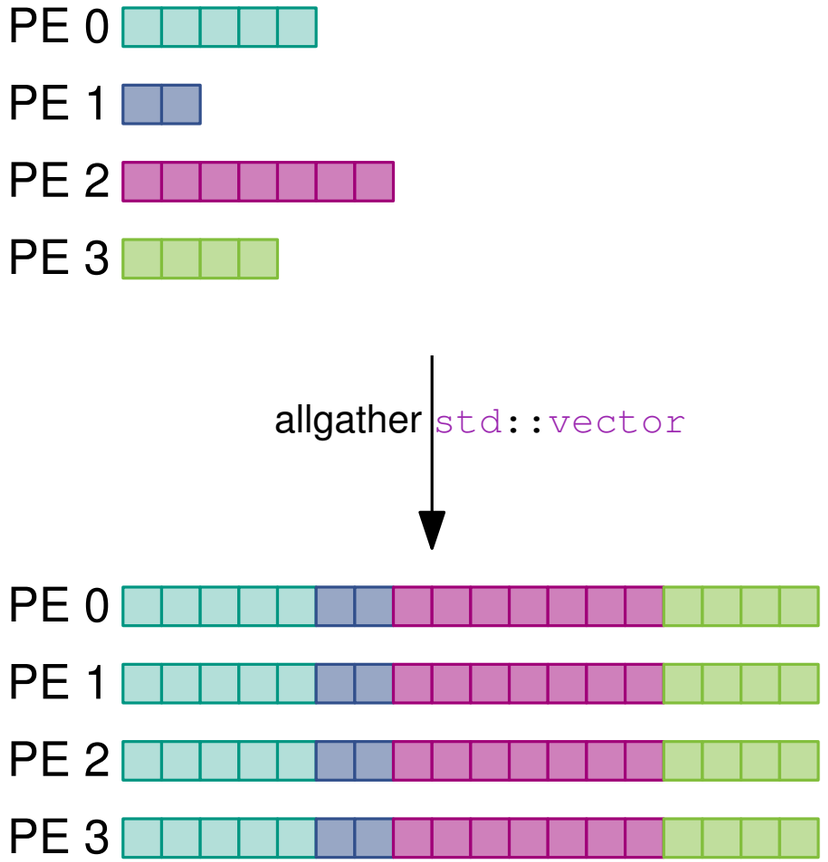




github.com/mschimek/scalable-distributed-string-sorting

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

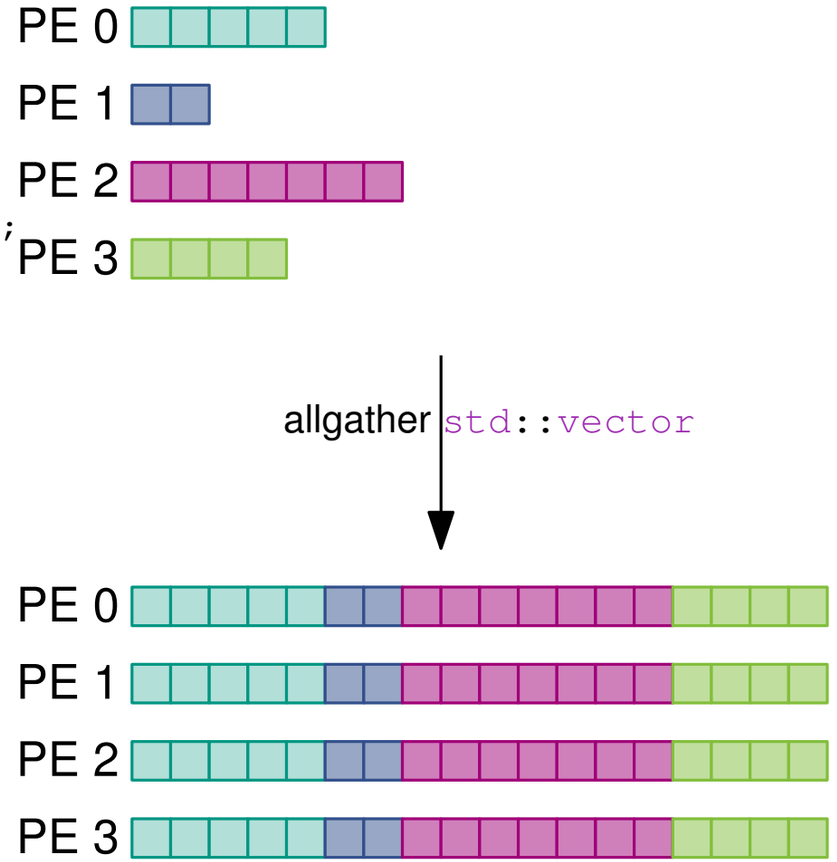
Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]



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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

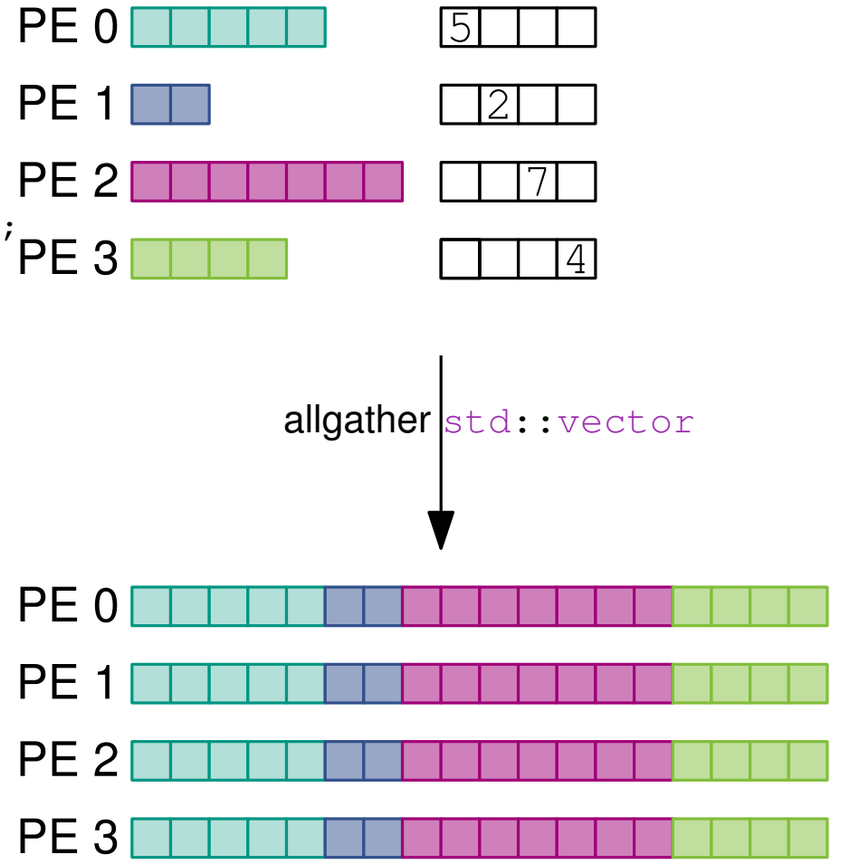
```cpp
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, 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;
}
```



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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

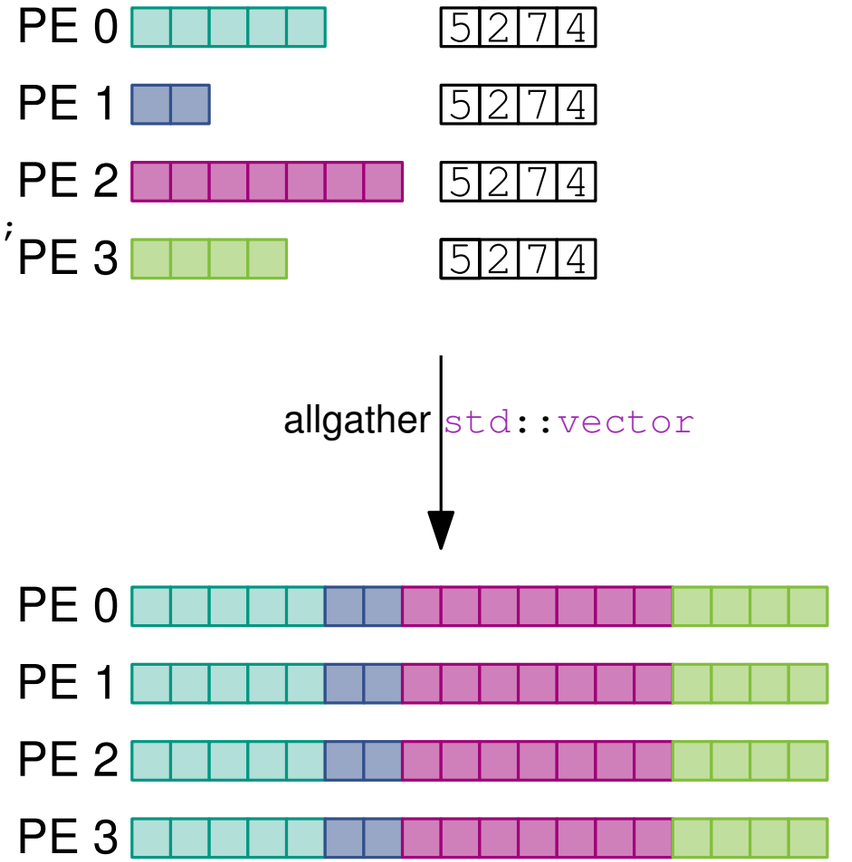
```cpp
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, 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;
}
```



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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

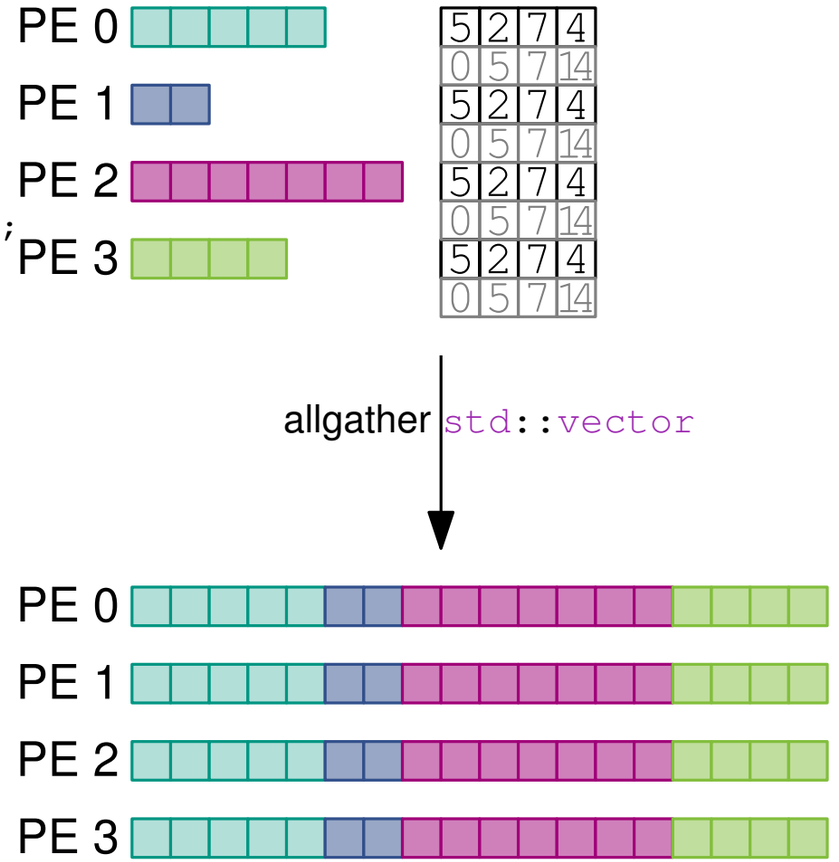
```cpp
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[rank] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, 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;
}
```



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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```cpp
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;
}
```



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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```cpp
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```cpp
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 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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```cpp
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 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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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** **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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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** **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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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** **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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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** **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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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 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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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** **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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```cpp
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));
}
```

```cpp
// 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:

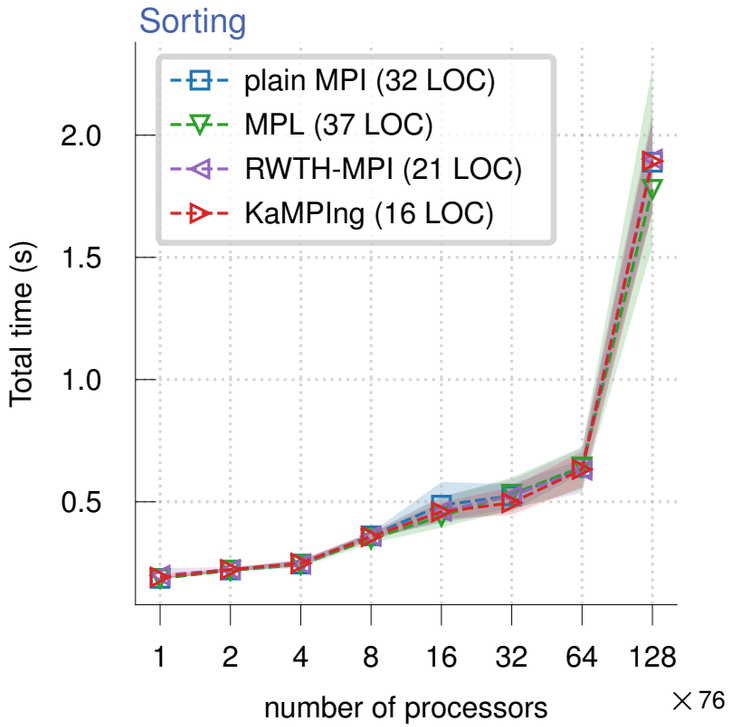
Karlsruhe **MPI** **n**ext **g**eneration

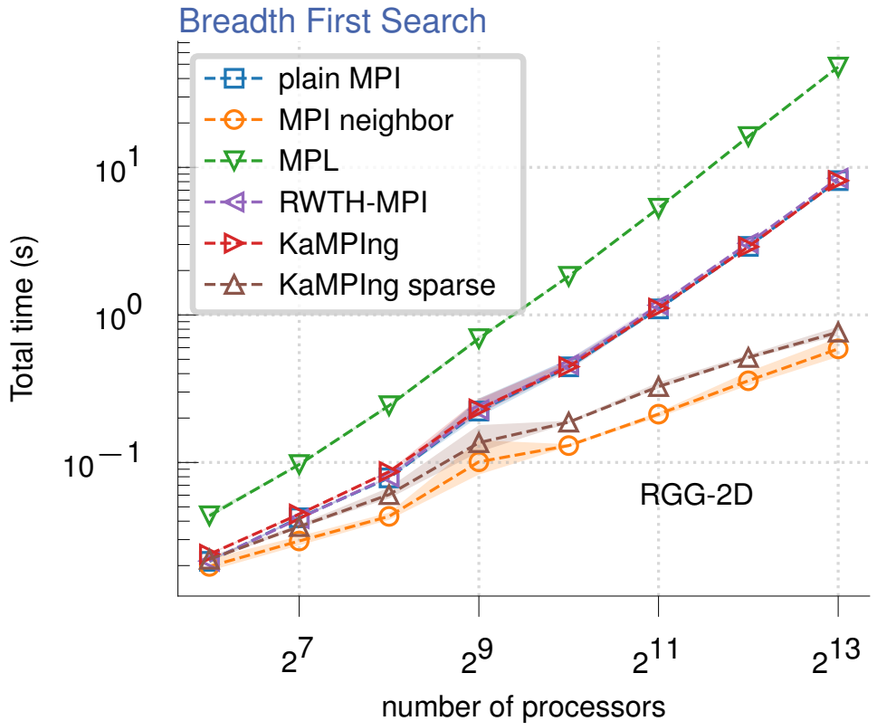
- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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:

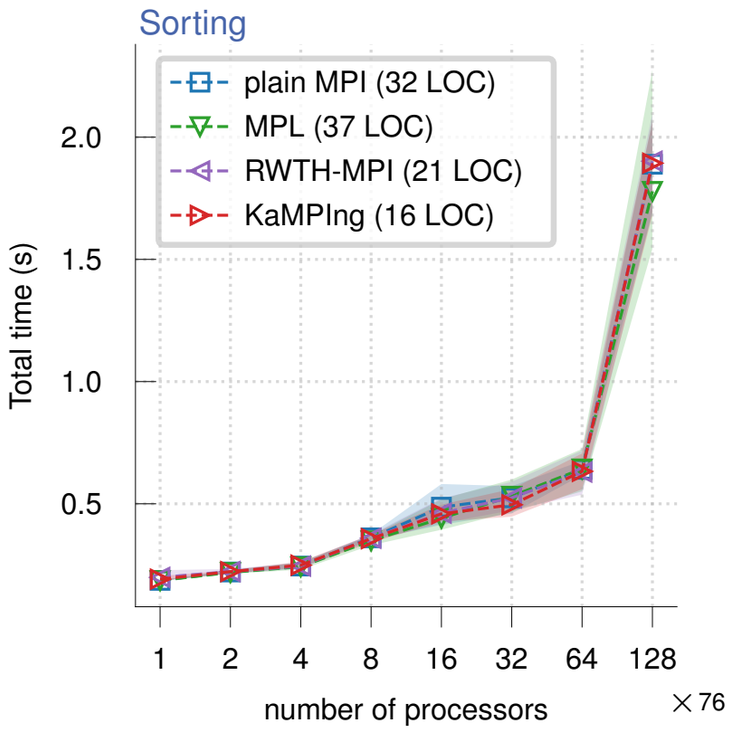
Karlsruhe **MPI** **n**ext **g**eneration

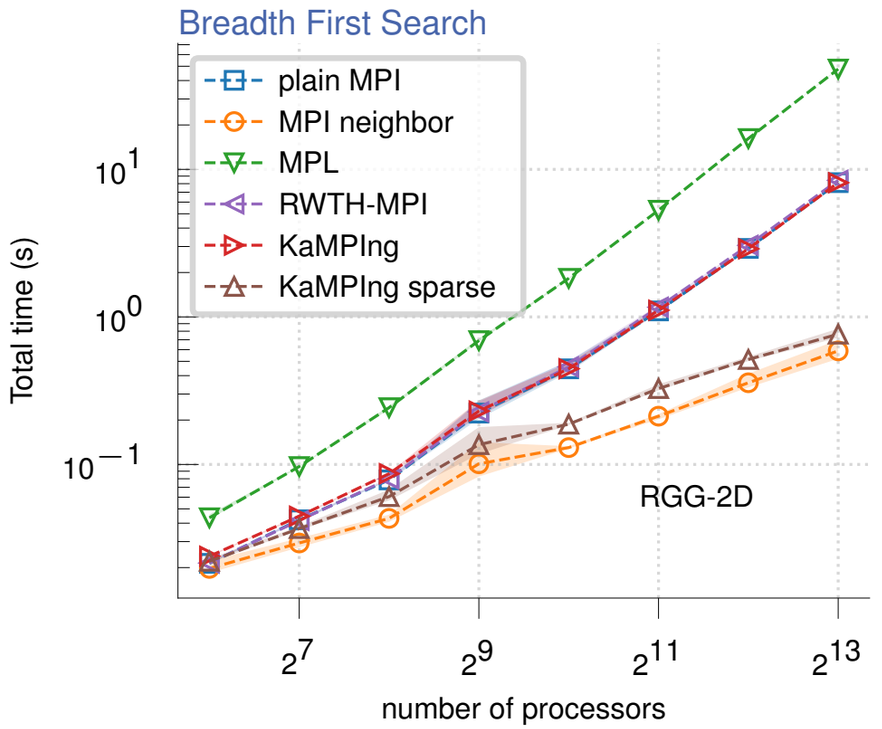
- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

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

Uhl, Schimek, Hübner, Kurpicz, Hespe, Seemaier, Sanders [SPAA'24, SC'24]

```
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));
}
```





github.com/kamping-site/kamping

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**

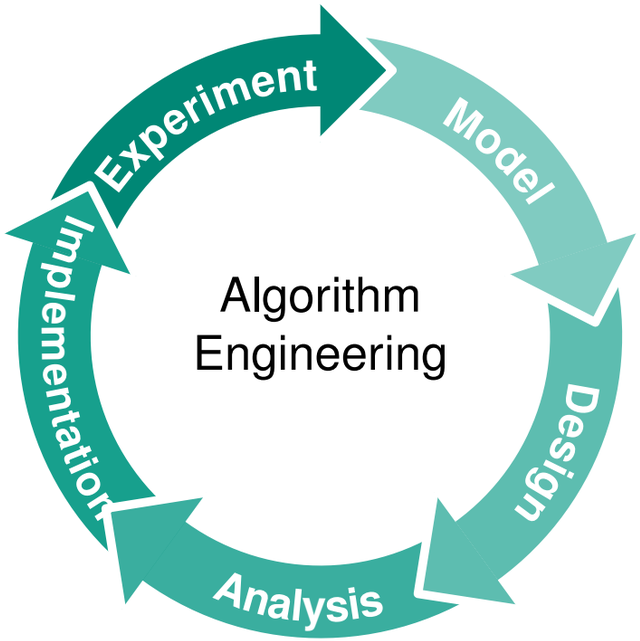
Conclusion

Our resource usage

- ≈ **3 million Core-h** used, currently CPU-only
- jobs differ from common HPC workloads
  - **irregular/skewed** communication patterns
  - short jobs at **large scale**
  - many parameter combinations
  - many iterative runs due to **algorithm engineering**

Our results

- state-of-the-art distributed **SAT solving system system**
- state-of-the-art distributed **graph partitioner**
- state-of-the-art distributed **string sorting**
- a new **C++ MPI library** for writing perfomant MPI code fast



European Research Council
Established by the European Commission

Dissertation Award

Best Reproducibility Advancement Award Finalist