MPI and OpenMP in Scientific Software Development

Day 2

Maxim Masterov
SURF, Amsterdam, The Netherlands
27-29.05.2024

Contents

Overview

Day 1

- **@16:00 Dr. Matthias Möller (TU Delft)**
- MPI (recap)
- Connecting to Snellius supercomputer
- Jacobi solver
- Performance analysis & debugging tools
- Hands-on

Day 2

- **@12:00 Dr. Nicola Spallanzani (MaX CoE)**
- MPI communications
- Domain decomposition
- MPI topologies
- Hands-on

Day 3

- **@12:00 Dr. Remco Havenith (RUG)**
- MPI IO
- Advanced OpenMP
- Hybrid programming
- Hands-on

Contents

Overview

Day 1

- **@16:00 Dr. Matthias Möller (TU Delft)**
- MPI (recap)
- Connecting to Snellius supercomputer
- Jacobi solver
- Performance analysis & debugging tools
- Hands-on

Day 2

- **@12:00 Dr. Nicola Spallanzani (MaX CoE)**
- MPI communications
- Domain decomposition
- MPI topologies
- Hands-on

Day 3

- **@12:00 Dr. Remco Havenith (RUG)**
- MPI IO
- Advanced OpenMP
- Hybrid programming
- Hands-on

MPI Communications

MPI Com

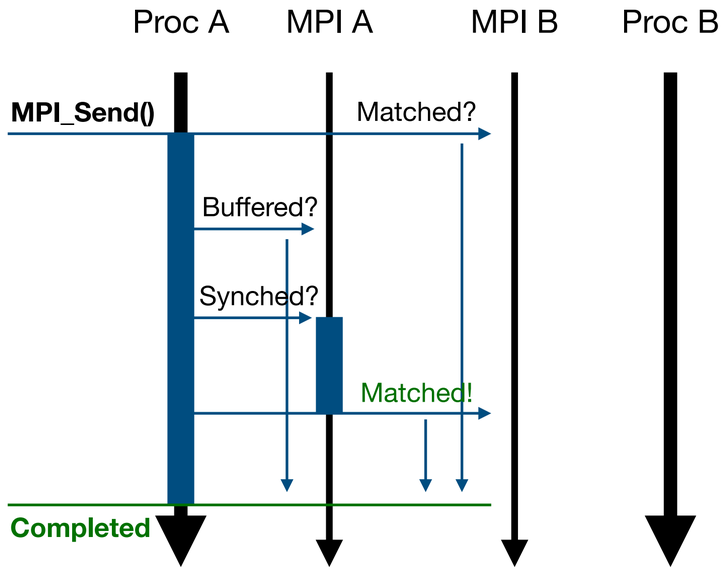
Sending (com

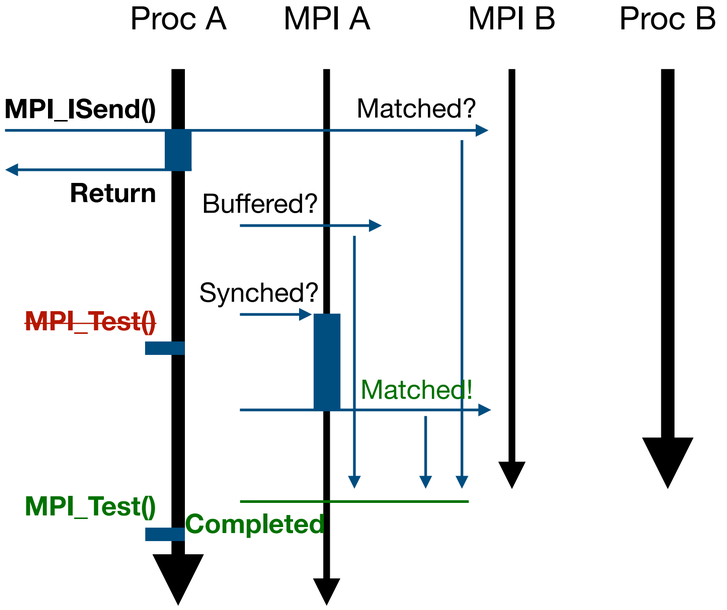
One of those scenarios may occur:
1. The message is directly passed to the receive buffer **(fast)**
2. The data is buffered (in temporary memory in the MPI implementation) **(fast)**
3. The function waits for a receiving process to appear **(slow)**

- **MPI_Send()** - doesn't return until the send buffer can be used. Depends on the message size it may or may not block, i.e. it may or may not wait until a matching receive is posted.

- **MPI_Isend()** - immediate send, but not necessarily asynchronous. This send must return to the user without requiring a matching receive at the destination. The send buffer **cannot** be reused until the send operation is finished (e.g. until successful **MPI_Wait()/ MPI_Test()**, or until the message is received).

- Always start with these functions. Move to fine-tuning functions only after your code is finished and validated.





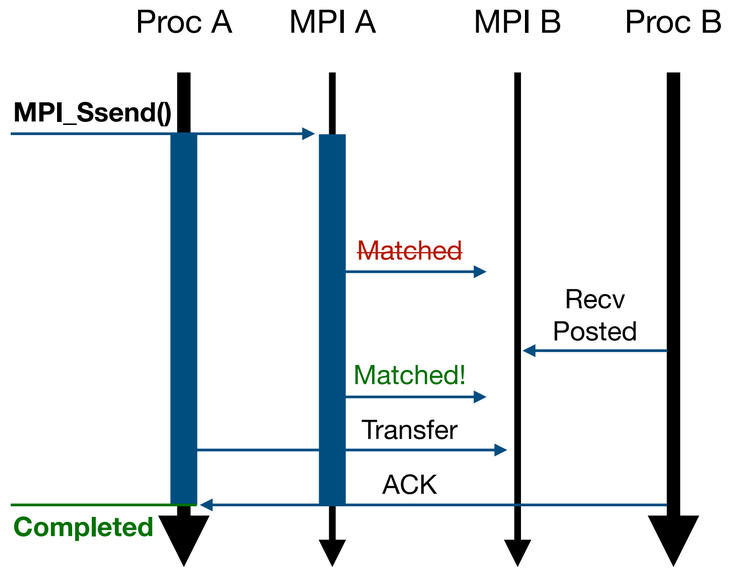
MPI Communications

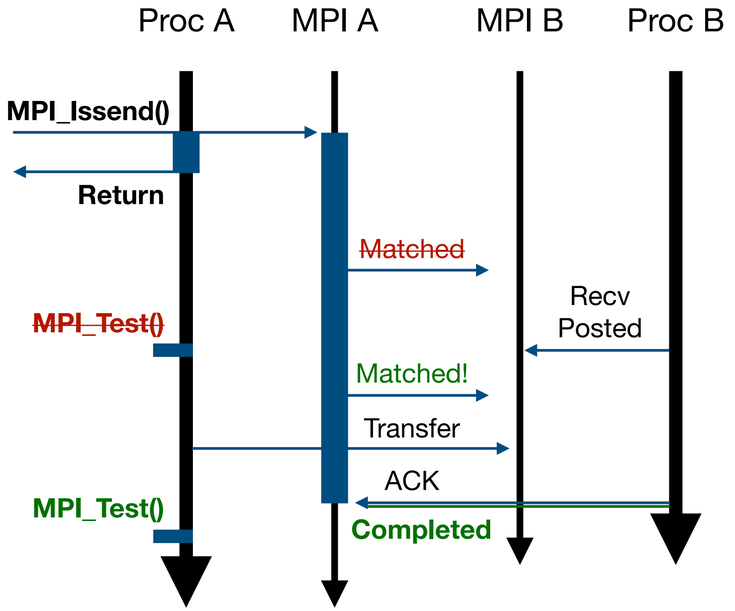
Sending (debugging)

- **MPI_Ssend()** - blocking synchronous send. It will not return until a matching receive is posted and the remote process has started receiving the message. The remote process sends *ACK* to the sender.

- **MPI_Issend()** - immediate nonblocking synchronous send. Similar to **MPI_Isend()**, but corresponding **MPI_Wait()/MPI_Test()** will complete only when the matching receive is posted.

- Should be used if the sender needs to know **when** the message is received. Allows to avoid local buffering. **MPI_Ssend()** can be used for debugging.





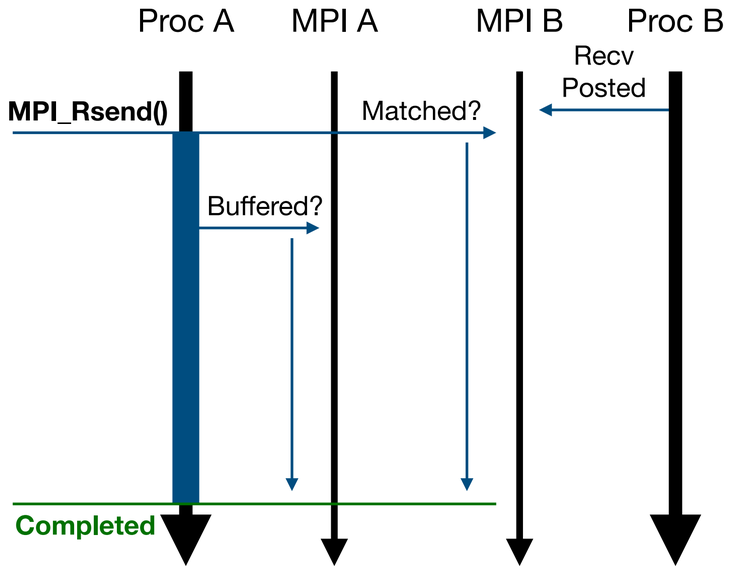
MPI Communications

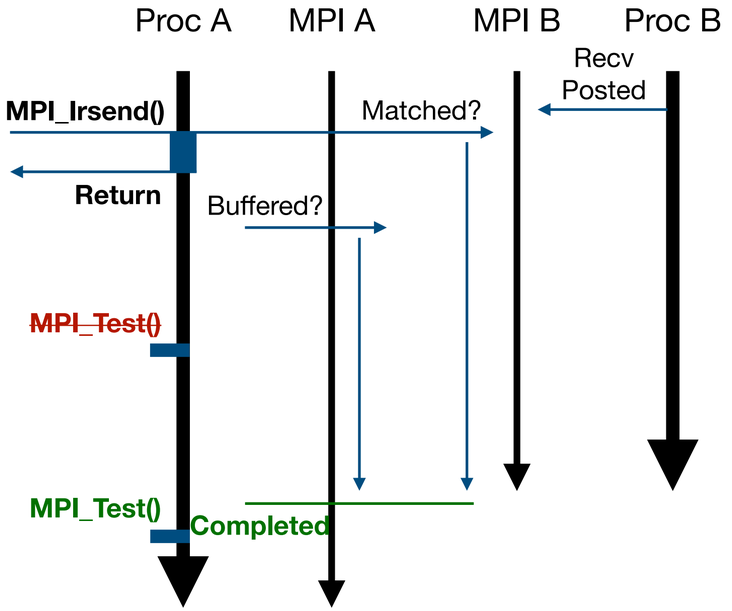
Sending (fine-tuning)

- **MPI_Rsend()** - blocking ready send. It may be used only if a matching receive is already posted (otherwise - erroneous). The developer is responsible for writing a correct program.

- **MPI_Irsend()** - immediate ready send.

```
{
    MPI_Recv(buf, size, MPI_DOUBLE, …);
    // …
    MPI_Rsend(buf, size, MPI_DOUBLE, …);
}
{
    MPI_Irecv(buf, size, MPI_DOUBLE, …);
    // …
    MPI_Irsend(buf, size, MPI_DOUBLE, …);
    // …
    MPI_Waitall(buf, size, MPI_DOUBLE, …);
}
```





MPI Communications

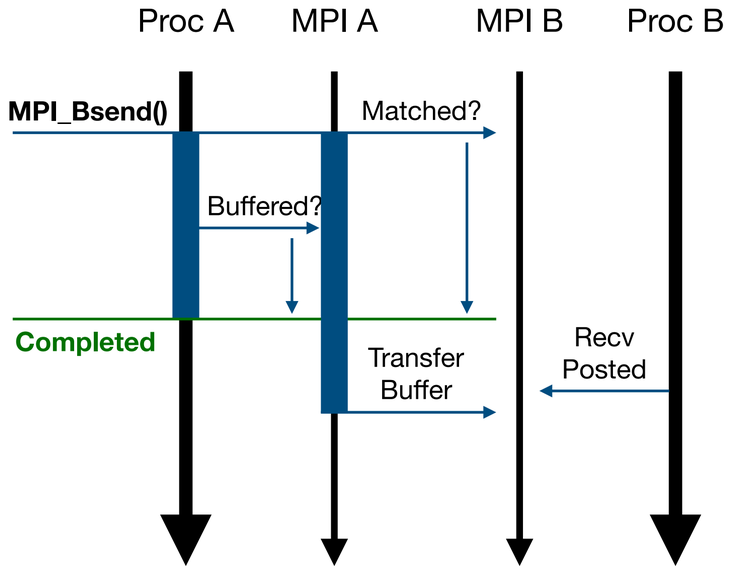
Sending (fine-tuning)

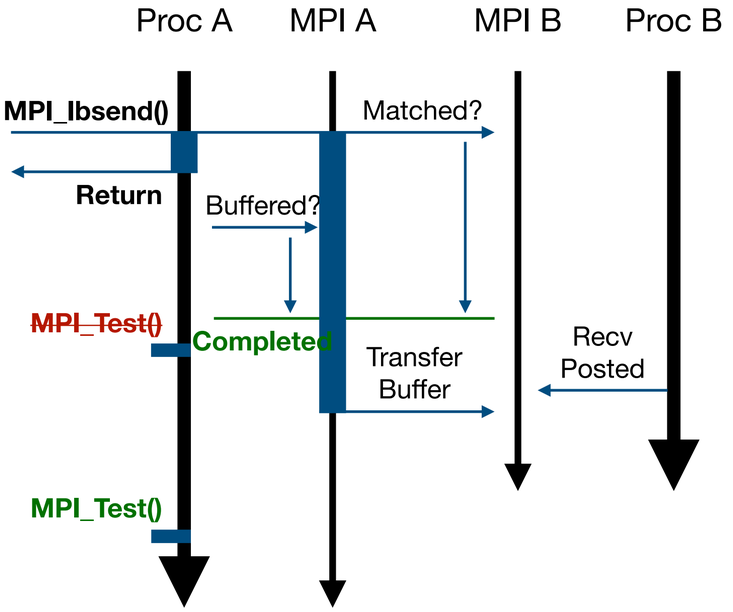
- **MPI_Bsend** - basic send with user-provided buffering. If there is no matching receive posted, the process will be blocked until the message is copied to the buffer. Therefore, the send buffer can be reused right after this function returns. Should be used only when absolutely necessary.

- **MPI_Ibsend** - nonblocking buffered send.

```
MPI_Buffer_attach(b, size*sizeof(double)
                  + MPI_BSEND_OVERHEAD);
for(int n = 0; n < N; ++n) {
    MPI_Bsend(buf, size, MPI_DOUBLE, …);
}
// …
MPI_Buffer_detach(b, n);
```

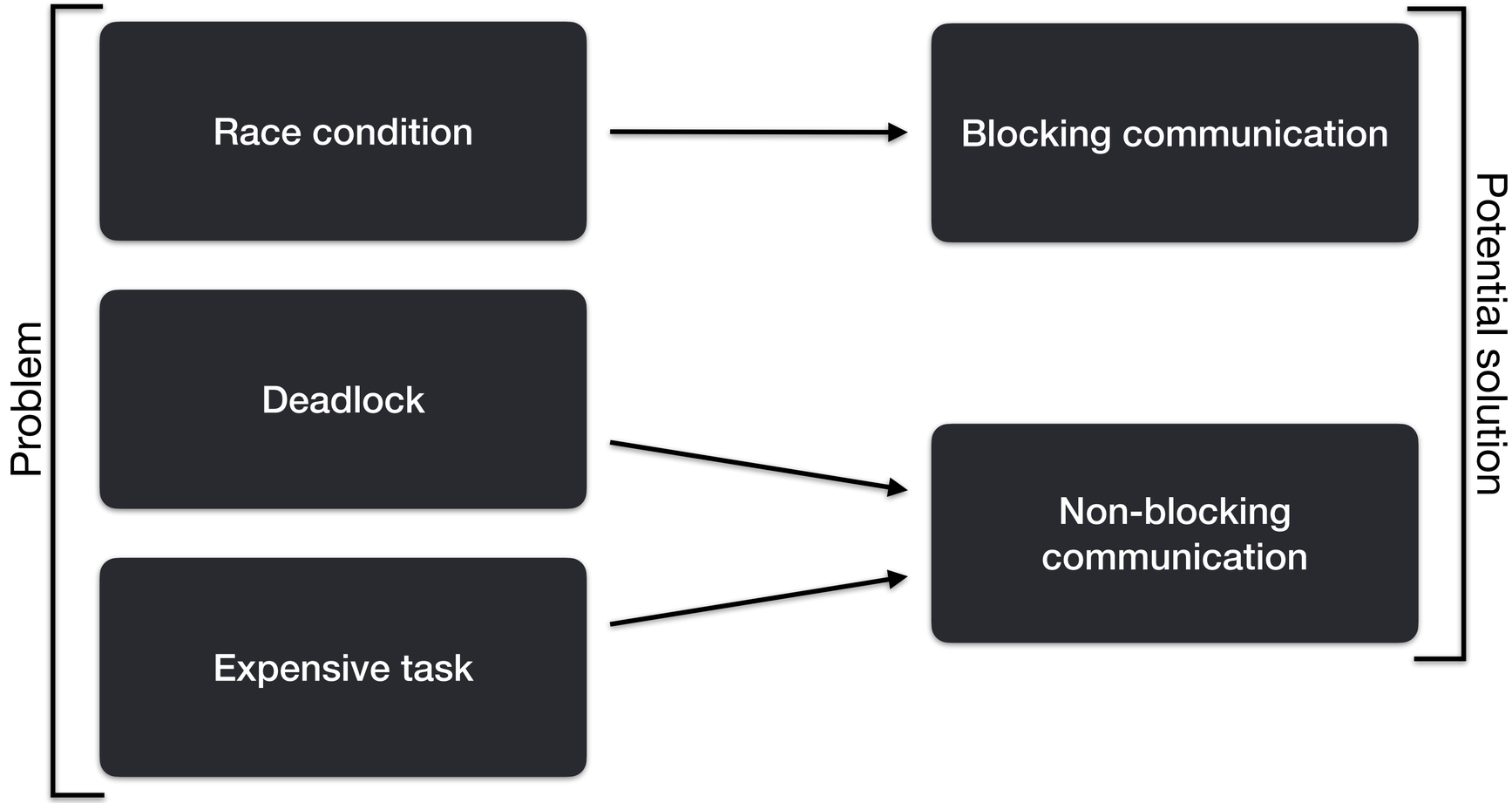
The user **MUST** provide the buffer space!





MPI Communications

Troubleshooting



Communications

Waiting

- **MPI_Wait()** waits for a non-blocking operation to complete. The call returns, in status, information on the completed operation. If the application does not need to examine the *status* field, one can save resources by using the predefined constant ***MPI_STATUS_IGNORE*** as a special value for the *status* argument.

```
/* Waits for an MPI request to complete */
int MPI_Wait(MPI_Request *request,                  // request
             MPI_Status *status);                   // status object (Status). May be MPI_STATUS_IGNORE

/* Tests for the completion of all previously initiated requests */
int MPI_Waitall(int count,                          // list length
                MPI_Request array_of_requests[],    // array of request handles
                MPI_Status *array_of_statuses)      // array of status objects (array of Status). May be
                                                    // MPI_STATUSES_IGNORE
```

Communications

Testing

- **MPI_Test()** checks if a non-blocking operation is complete at a given time. In such a case, the status object is set to contain information on the completed operation. Unlike **MPI_Wait()**, **MPI_Test()** will not wait for the underlying non-blocking operation to complete. If the application does not need to examine the *status* field, one can save resources by using the predefined constant ***MPI_STATUS_IGNORE*** as a special value for the *status* argument

```
/* Tests for the completion of a request */
int MPI_Test(MPI_Request *request,  // MPI request
             int *flag,             // true if operation completed
             MPI_Status *status);   // status object (Status). May be MPI_STATUS_IGNORE

/* Tests for the completion of all previously initiated requests */
int MPI_Testall(int count,                          // lists length
                MPI_Request array_of_requests[],    // array of requests
                int *flag,                          // True if all requests have completed; false otherwise
                MPI_Status array_of_statuses[]);    // array of status objects (array of Status). May be
                                                    // MPI_STATUSES_IGNORE
```

Communications

Probing

- The message can be “probed” without the actual receive. The developer can then decide how to receive the message, e.g. by deciding the receive buffer size
- **MPI_Probe()** is blocking, whereas **MPI_Iprobe()** is immediate. Other than that, their behaviour is identical

```
/* Blocking test for a message */
int MPI_Probe(int source,           // Source rank or MPI_ANY_SOURCE
              int tag,              // Tag value or MPI_ANY_TAG
              MPI_Comm comm,        // Communicator
              MPI_Status *status);  // Status object


/* Nonblocking test for a message */
int MPI_Iprobe(int source,          // Source rank or MPI_ANY_SOURCE
               int tag,             // Tag value or MPI_ANY_TAG
               MPI_Comm comm,       // Communicator
               int *flag,           // True if a message with the specified
                                    // source, tag, and communicator is
                                    // available
               MPI_Status *status); // Status object
```

This structure contains all essential information

Dynamic receiving

MPI_Status

The rank of the sender

The tag of the sender

The error code

MPI_SUCCESS - no error
MPI_ERR_COUNT - invalid count
MPI_ERR_TYPE - invalid datatype
MPI_ERR_IN_STATU - error code is in
MPI_ERR_BUFFER - invalid buffer pointer
...

```
/* ompi/include/mpi.h */

…

typedef struct ompi_status_public_t MPI_Status;

…

/*
 * MPI_Status
 */
struct ompi_status_public_t {
    /* These fields are publicly defined in the
       MPI specification. User applications may
       freely read from these fields. */
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
    /* The following two fields are internal to the
       Open MPI implementation and should not be
       accessed by MPI applications. They are subject
       to change at any time.  These are not the
       droids you're looking for. */
    int _cancelled;
    size_t _ucount;
};
```

Communications

Probing: unknown size

Only the root process knows the message size

MPI_Probe() is blocking! It doesn't receive. The sender ID can be found in status: **MPI_SOURCE**

Returns the message size

Allocate memory and receive the message as usual

```
if (my_rank == root_proc) {
    // Send the same message to all processes
    for(int proc_id = 0; proc_id < num_proc; ++proc_id) {
        if (my_rank != proc_id)
            MPI_Send(send_data,
                     msg_size,
                     MPI_CHAR,
                     proc_id,
                     tag,
                     loc_mpi_comm);
    }
}
else {
    int recv_size = 0;
    char *recv_data;

    // Probe the message size and allocate memory
    MPI_Probe(root_proc,
              tag,
              loc_mpi_comm,
              &status);

    // Get the size
    MPI_Get_count(&status,
                  MPI_CHAR,
                  &recv_size);


    // Pre-allocate memory and receive
    recv_data = new char [recv_size];
    MPI_Recv(recv_data,
             recv_size,
             MPI_CHAR,
             root_proc,
             tag,
             loc_mpi_comm,
             &status);
    ...
    delete [] recv_data;
}
```

Communications

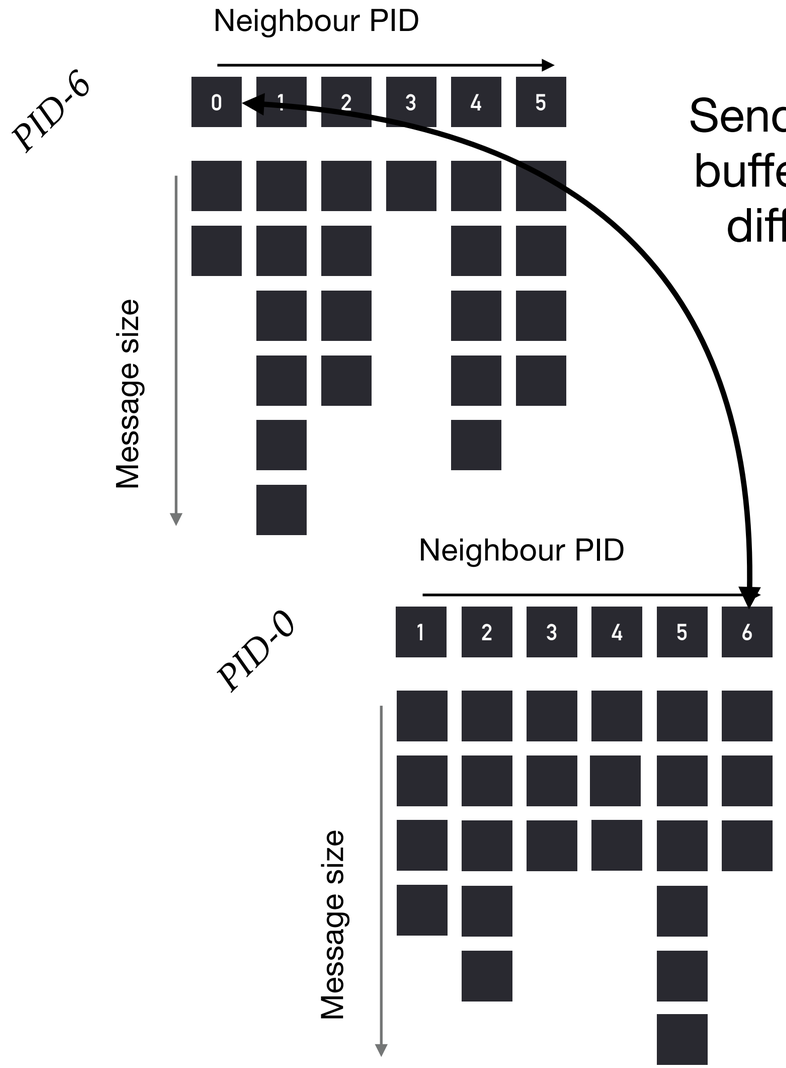
Probing: unknown sender

- Some processes send an integer to other processes, but we do not know who receives from whom
- All processes may expect the message (even the sender)
- **MPI_Iprobe()** doesn't receive, it only probes
- Use **MPI_ANY_TAG** If the tag is unknown
- **MPI_Status** contains all necessary information

```
int total_sends = 0;
for(; it_i != it_i_end; ++it_i) {
    int pid = it_i->first;
    MPI_Isend(&it_i->second.size(), 1, MPI_INT, pid,
              tag, loc_mpi_comm,
              rqst_arr[total_sends++]);
}

MPI_Allreduce(MPI_IN_PLACE, total_sends, …,
              MPI_SUM, …);

int total_receives = 0;
while(1) {
    int flag = false;
    MPI_Status status;

    MPI_Iprobe(MPI_ANY_SOURCE, tag, loc_mpi_comm,
               &flag, &status);

    if (flag == 1) {
        int sender_id = status.MPI_SOURCE;
        int num_remote_points = 0;
        MPI_Status status_rcv;
        MPI_Recv(&num_remote_points, 1, MPI_INT,
                 sender_id, tag, loc_mpi_comm,
                 &status_rcv);
        ++total_receives;
        // Do something with num_remote_points…
    }
    MPI_Allreduce(MPI_IN_PLACE, total_receives, …,
                  MPI_SUM, …);
    if (total_receives == total_sends)
        break;
}

…

MPI_Waitall(…);
```

Communications

Sending buffers of different sizes



Send and receive buffers can be of different sizes

Use “snd” map

Use “rcv” map

```
// assemble send messages using data from real cells
for(/* all processes in the neighbourhood */) {
    for(int n = 0; n < msg_size; ++n) {
        /* assemble an array of data to be sent */
    }
}

// iterate over neighbours and send data
int counter = 0;
for(/* all processes in the neighbourhood */) {
    int proc_id = /* PID from the neighbourhood */;
    MPI_Isend(send_data, msg_size, _data_type,
              proc_id, tag,
              local_mpi_comm, &request_snd[counter]);
    ++counter;
}
```

```
// iterate over neighbors and receive data
counter = 0;
for(/* all processes in the neighbourhood */) {
    int proc_id = /* PID from the neighbourhood */;
    MPI_Irecv(recv_data, msg_size, _data_type,
              proc_id, tag, local_mpi_comm,
              &request_rcv[counter]);
    ++counter;
}

MPI_Waitall(num_proc_in_ngb, request_snd, status_snd);
MPI_Waitall(num_proc_in_ngb, request_rcv, status_rcv);

// disassemble messages and assign data to halo cells
for(/* all processes in the neighbourhood */) {
    int proc_id = /* PID from the neighbourhood */;
    for(int n = 0; n < msg_size; ++n) {
        /* disassemble an array of data to be sent */
    }
}
```

Non-blocking communications

Trap 1

- **MPI_Wait(all)** should be called by a process that originally invoked a non-blocking communication!

```
if (getMyRank() == root_pid) {
    for (n = 0..N && n != root_pid)
        MPI_Isend(…,
                  &request_arr[n]);
}
MPI_Waitall(num_procs,
            request_arr,
            status_arr);
```

```
if (getMyRank() == root_pid) {
    for (n = 0..N && n != root_pid)
        MPI_Isend(…,
                  &request_arr[n]);
}

if (getMyRank() == root_pid) {
    MPI_Waitall(num_procs,
                request_arr,
                status_arr);
}
```

Non-blocking communications

Trap 2

- If **MPI_Isend()** is used, the buffer cannot be reused or destroyed until the request is returned!

```
if (getMyRank() == root_pid) {
    std::vector<Field> snd_field(num_procs);
    …
    MPI_Isend(snd_field,
              …);
}
MPI_Recv(…,
         root_pid,
         …);
```

```
std::vector<Field> snd_field(num_procs);
if (getMyRank() == root_pid) {
    …
    MPI_Isend(snd_field,
              …);
}
MPI_Recv(…,
         root_pid,
         …);
```

Hands-on #2.1

Hands-on

Immediate communications

- Improve the real-halo data exchange in the Jacobi code (**DataTypes/vector.cpp**):
  - Using non-blocking send
  - Using non-blocking send and receive
- Profile the code:
  - 1-16 cores
  - Use **“-s 64 64 -d 1 #cores”** options for the executable
- Optimise calls for the **calculateNorm()** function

```
int MPI_Isend(const void *buf,          // initial address
                                        // of send buffer
              int count,                // number of
                                        // elements in
                                        // send buffer
              MPI_Datatype datatype,    // datatype of
                                        // each send
                                        // buffer element
              int dest,                 // rank of
                                        // destination
              int tag,                  // message tag
              MPI_Comm comm,            // communicator
              MPI_Request *request);    // communication
                                        // request
int MPI_Irecv(void *buf,                // initial address
                                        // of receive buffer
              int count,                // number of elements
                                        // in receive buffer
              MPI_Datatype datatype,    // datatype of each
                                        // receive buffer
                                        // element
              int source,               // rank of source
              int tag,                  // message tag
              MPI_Comm comm,            // communicator
              MPI_Request *request);    // communication
                                        // request
```

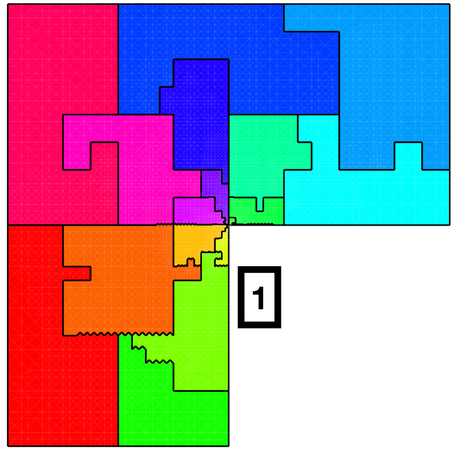
@12:00
Dr. Nicola Spallanzani, MaX CoE
“Materials design at the eXascale: performance portability of the Yambo code with deviceXlib”

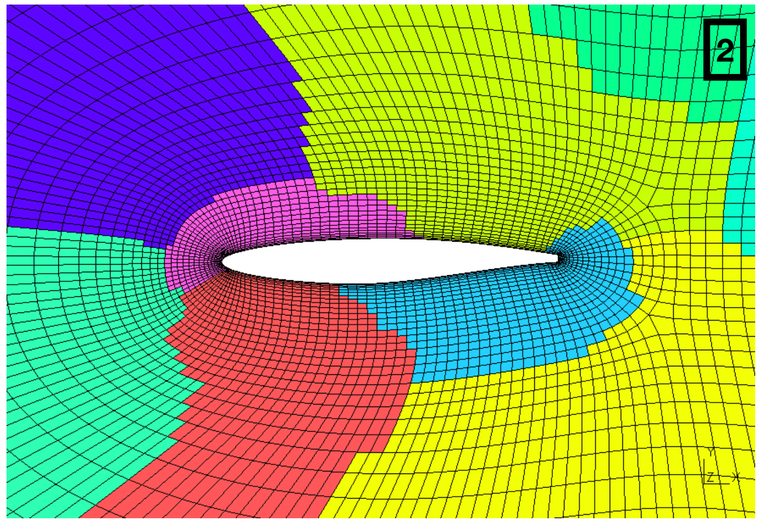
Domain Decomposition

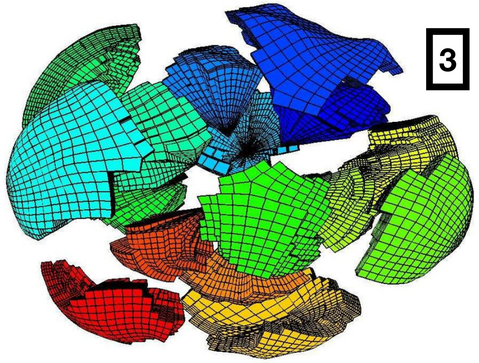
Domain decomposition

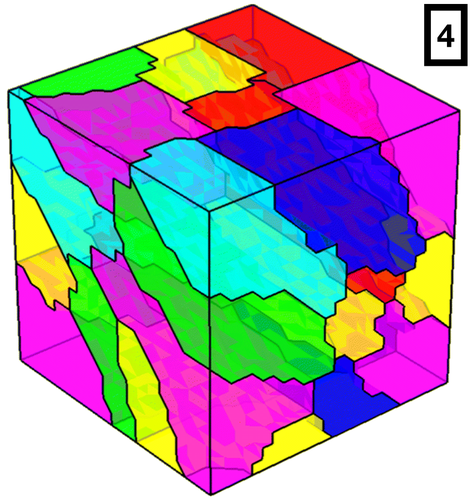
Decomposition Types

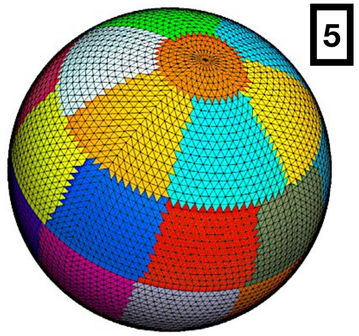
- Unstructured grids and graphs:
  - METIS, ParMETIS
  - Scotch, PT_Scotch
  - Zoltan
- Main issues:
  - Surface area reduction
  - Edges cutting reduction

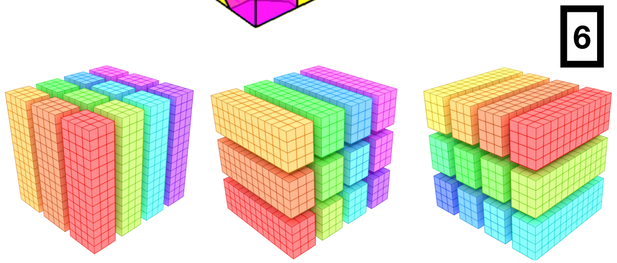










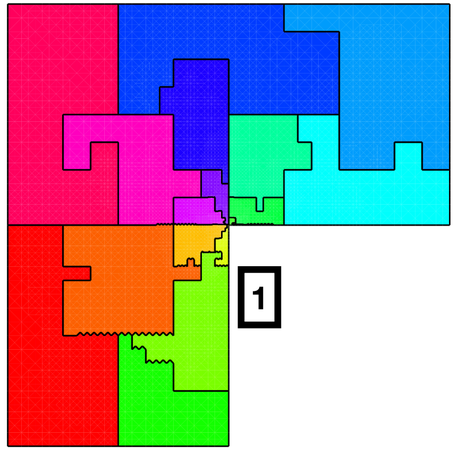


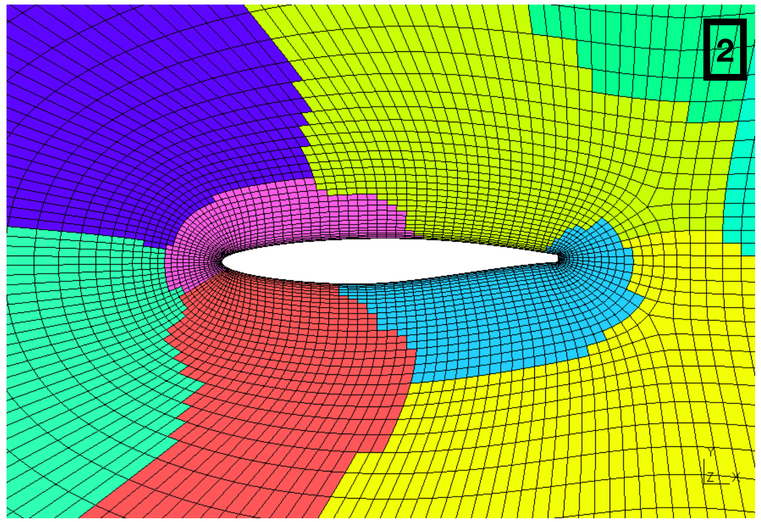
1. Mitchell, W.F., A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids , J. Par. Dist. Comp., 67 (4), 2007, pp. 417-429
2. Nieplocha, Jarek & Harrison, Robert & Kumar, Mukul & Palmer, Bruce & Tipparaju, Vinod & Trease, Harold. (2002). Combining Distributed and Shared Memory Models: Approach and Evolution of the Global Arrays Toolkit.
3. https://nutscfd.wordpress.com/2017/03/06/mesh-partitioning-using-parmetis/
4. Calvo, Juan G. (2020) A new coarse space for overlapping Schwarz algorithms for H(curl) problems in three dimensions with irregular subdomains. Numerical Algorithms. 83, 885–899
5. Deconinck, Willem & Bauer, Peter & Diamantakis, Michail & Hamrud, Mats & Kühnlein, Christian & Maciel, Pedro & Mengaldo, Gianmarco & Quintino, Tiago & Raoult, Baudouin & Smolarkiewicz, Piotr & Wedi, Nils. (2017). Atlas : A library for numerical weather prediction and climate modelling. Computer Physics Communications.
6. Verma, M.K., Samuel, R., Chatterjee, S. *et al.* (2020). Challenges in Fluid Flow Simulations Using Exascale Computing. *SN COMPUT. SCI.* **1,** 178

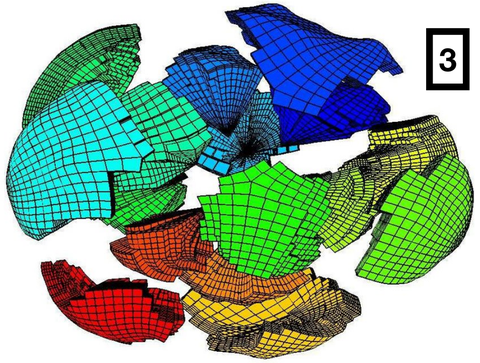
Domain decomposition

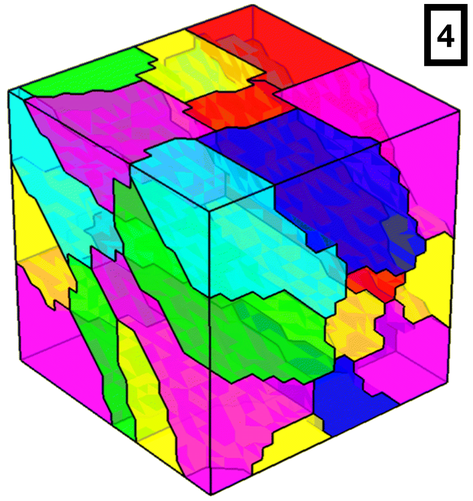
Decomposition Types

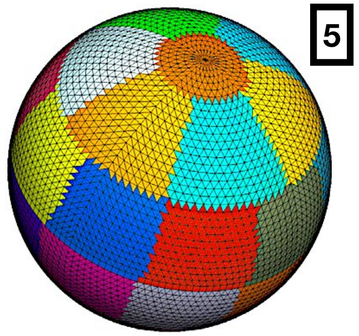
- **Unstructured grids and graphs:**
  - METIS, ParMETIS
  - Scotch, PT_Scotch
  - Zoltan
- **Maps:**
  - **Global-to-local cells**
  - **Real-to-halo and halo-to-real**

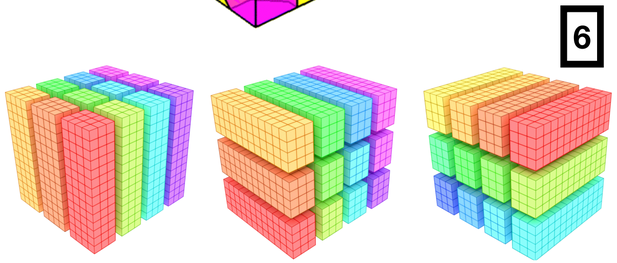










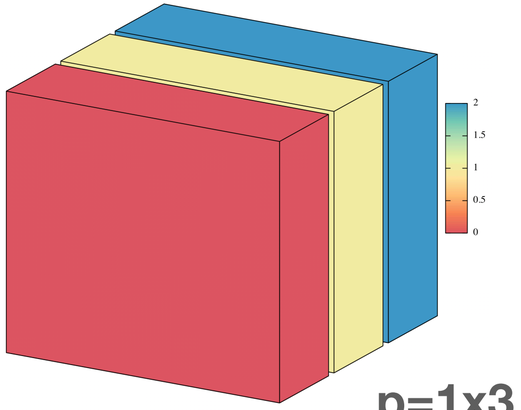


1. Mitchell, W.F., A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids , J. Par. Dist. Comp., 67 (4), 2007, pp. 417-429
2. Nieplocha, Jarek & Harrison, Robert & Kumar, Mukul & Palmer, Bruce & Tipparaju, Vinod & Trease, Harold. (2002). Combining Distributed and Shared Memory Models: Approach and Evolution of the Global Arrays Toolkit.
3. https://nutscfd.wordpress.com/2017/03/06/mesh-partitioning-using-parmetis/
4. Calvo, Juan G. (2020) A new coarse space for overlapping Schwarz algorithms for H(curl) problems in three dimensions with irregular subdomains. Numerical Algorithms. 83, 885–899
5. Deconinck, Willem & Bauer, Peter & Diamantakis, Michail & Hamrud, Mats & Kühnlein, Christian & Maciel, Pedro & Mengaldo, Gianmarco & Quintino, Tiago & Raoult, Baudouin & Smolarkiewicz, Piotr & Wedi, Nils. (2017). Atlas : A library for numerical weather prediction and climate modelling. Computer Physics Communications.
6. Verma, M.K., Samuel, R., Chatterjee, S. *et al.* (2020). Challenges in Fluid Flow Simulations Using Exascale Computing. *SN COMPUT. SCI.* **1,** 178

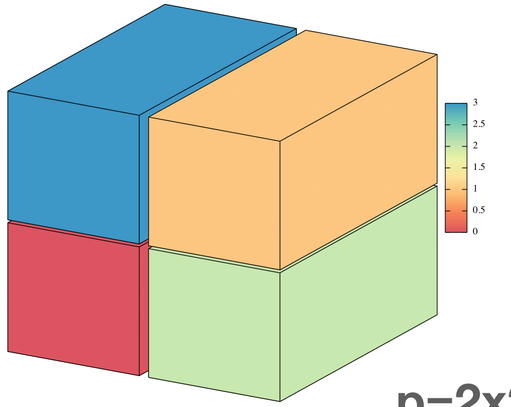
Domain decomposition

Decomposition Types

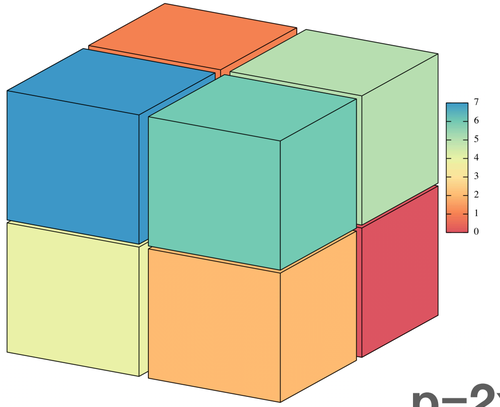
- Structured decomposition is very simple, but has its limitations
  - Restricted to the number of specified processes
  - Might not lead to the optimal load balance



p=1x3



p=2x2

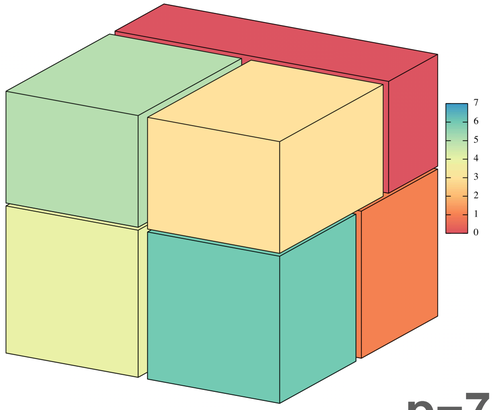


p=2x2x2

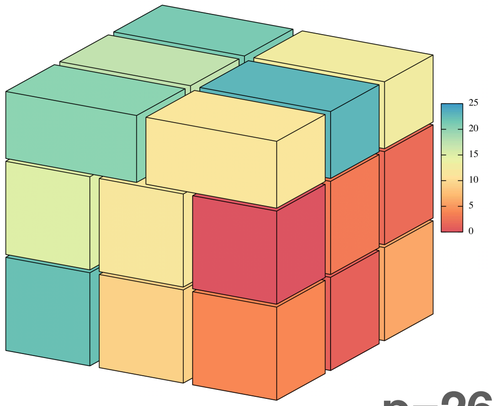
Domain decomposition

Decomposition Types

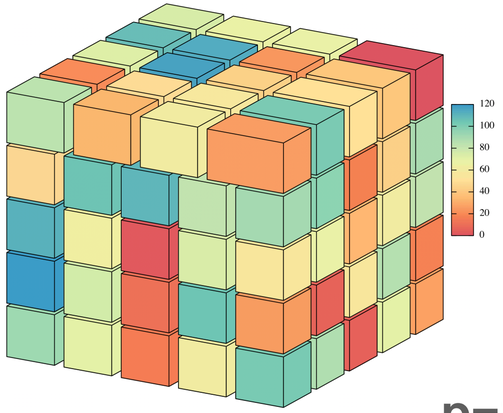
- Even structured grids can be decomposed using unstructured decomposition
- This allows to create more flexible and generic codes
- But increases complexity



p=7



p=26

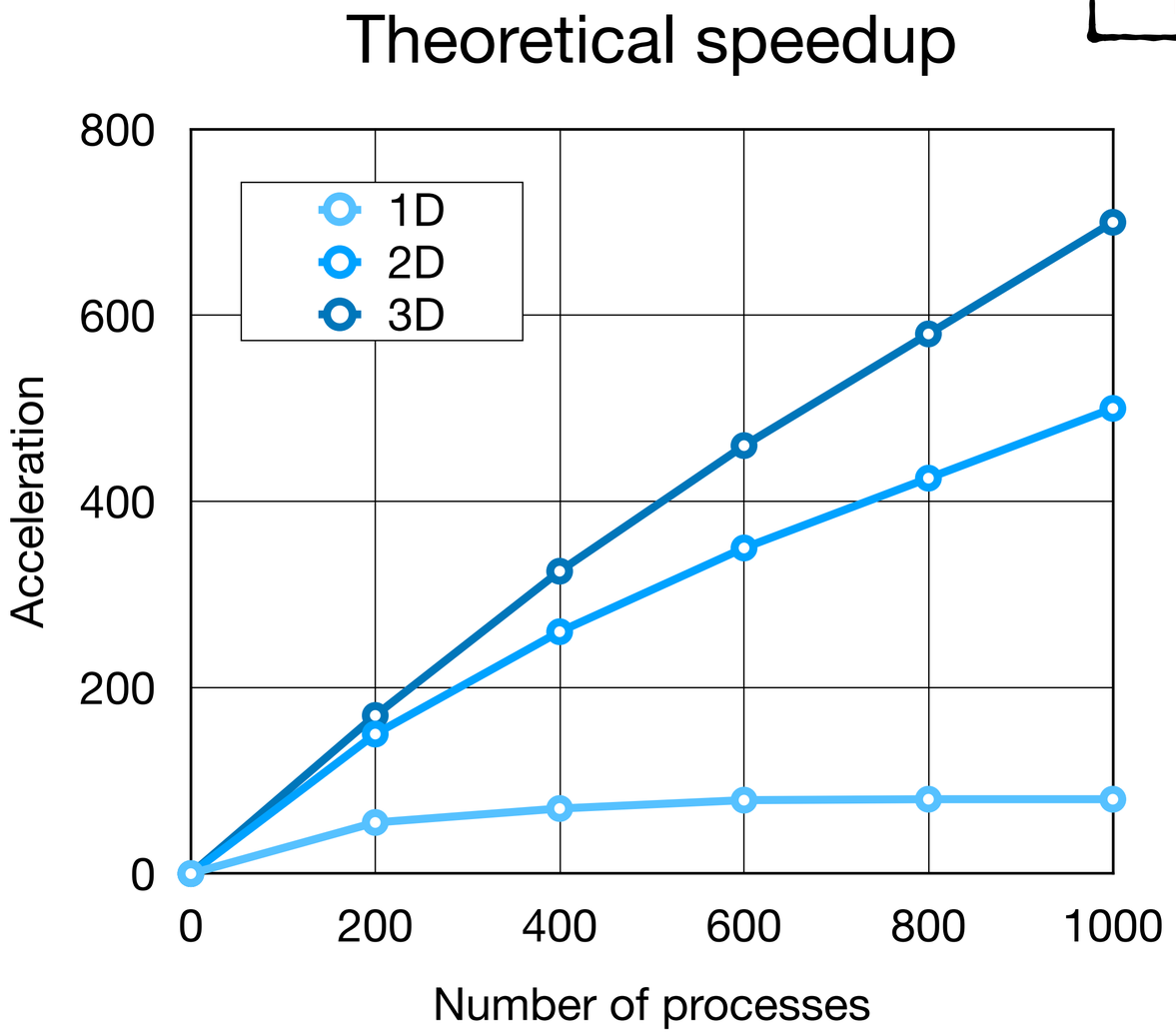


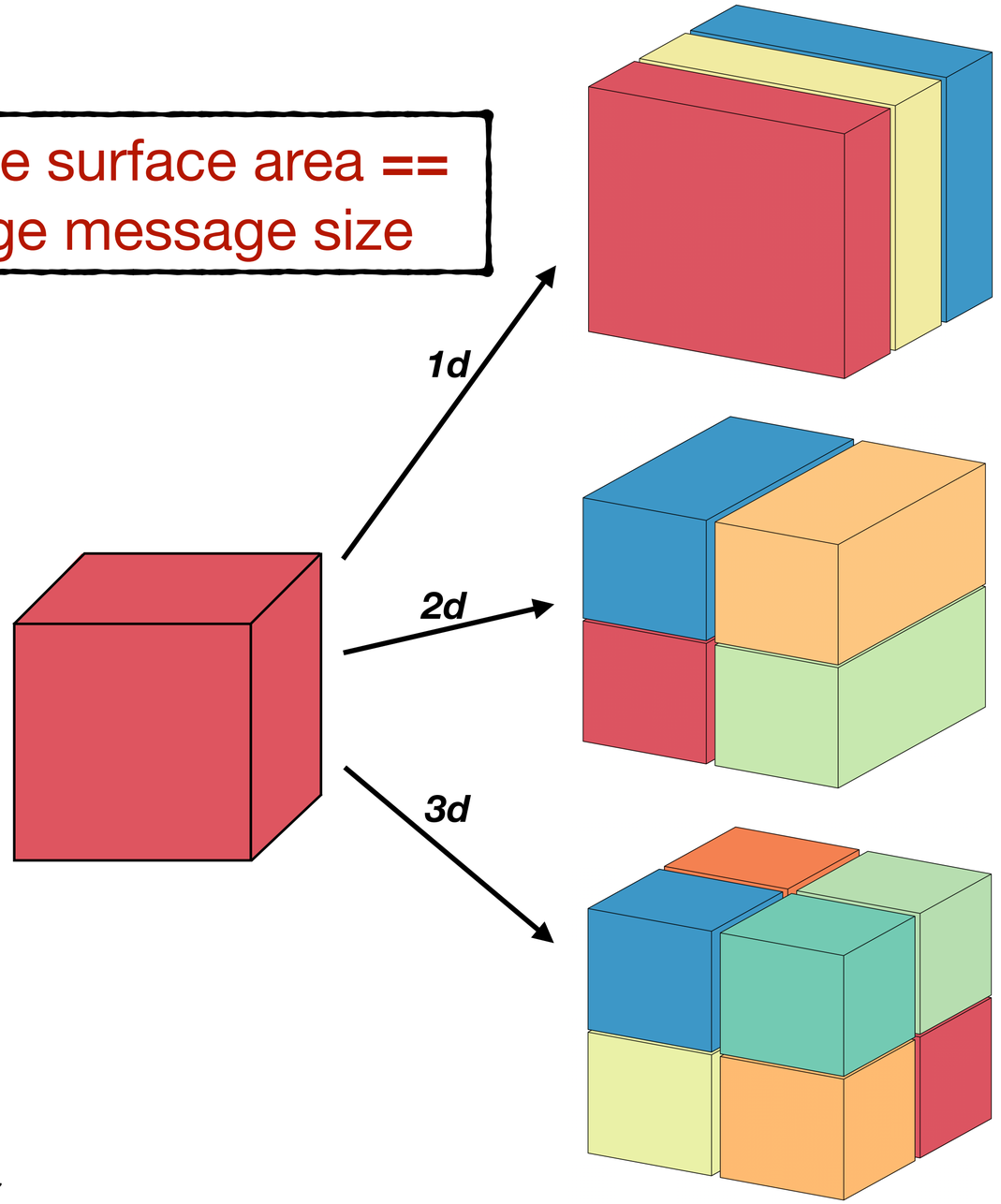
p=120

Domain decomposition

Decomposition Types

Large surface area == large message size





Hands-on #2.2

Hands-on

Influence of the decomposition

- Check the influence of the domain decomposition for the Jacobi solver
  - use **100x100** cells.
  - compile code with the “***mpi”*** type
  - execute on one node with 1..128 cores
  - change the decomposition flag ***-d*** and compare 1D to 2D decomposition
- For instance, try the following options:
  - 1D: -d 1 1; -d 1 2; -d 1 4; -d 1 8; -d 1 16;
  - 2D: -d 2 2; -d 2 4; -d 4 4;

10K iteration of the Jacobi solver on a single EPYC node (2x64), Snellius

Num procs.	1D		2D	
	time, [s]	-d	time, [s]	-d
1				
2				
4				
8				
16				
32				
64				
128				

Hands-on

Influence of the decomposition

- Check the influence of the domain decomposition for the Jacobi solver
  - use **100x100** cells.
  - compile code with the "***mpi"*** type
  - execute on one node with 1..16 cores
  - change the decomposition flag ***-d*** and compare 1D to 2D decomposition
- For instance, try the following options:
  - 1D: -d 1 1; -d 1 2; -d 1 4; -d 1 8; -d 1 16;
  - 2D: -d 2 2; -d 2 4; -d 4 4;

Huge difference in performance!

10K iteration of the Jacobi solver on a single EPYC node (2x64), Snellius

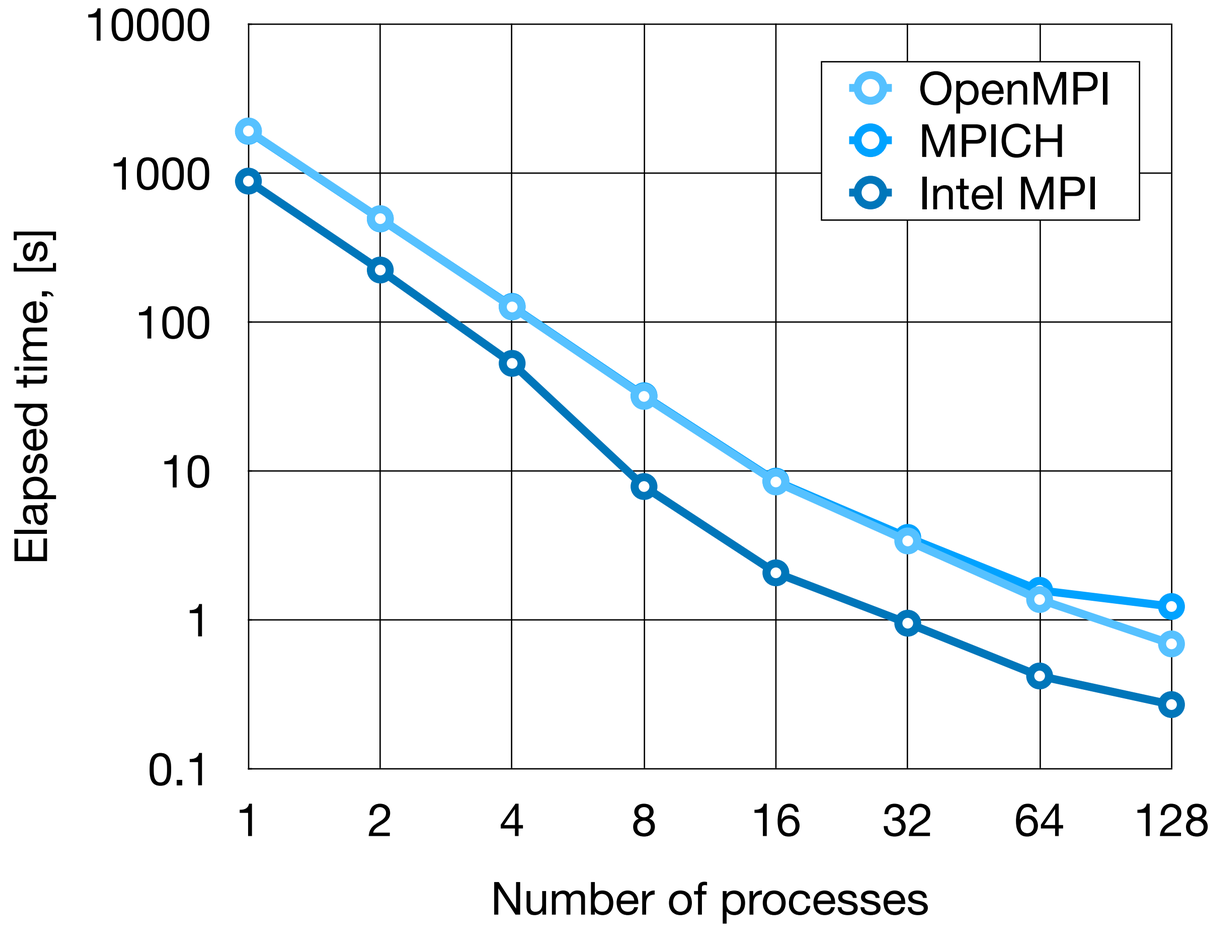
Num procs.	1D		2D	
	time, [s]	-d	time, [s]	-d
1	1918.75	1x1	1918.75	1x1
2	494.46	1x2	494.46	1x2
4	130.39	1x4	126.25	2x2
8	52.97	1x8	31.66	2x4
16	20.49	1x16	8.44	4x4
32	10.26	1x32	3.39	4x8
64	**275.28**	**1x64**	**1.37**	**8x8**
128	**-**	**1x128**	**0.69**	**8x16**

Why the speedup is ~4?

Hands-on

Influence of the decomposition

- Recompile the code using **Intel MPI** and **MPICH**
- Re-run tests using 2D decomposition
- Compare the performance of different MPI implementations
- Any ideas why Intel MPI is faster?



Domain Decomposition

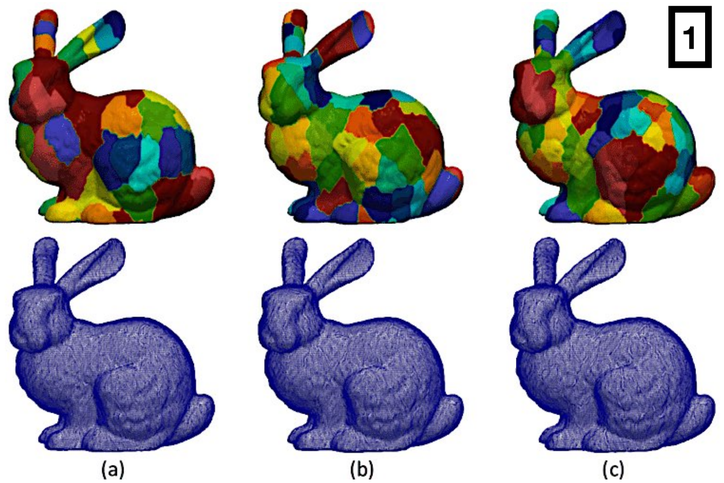
Domain decomposition

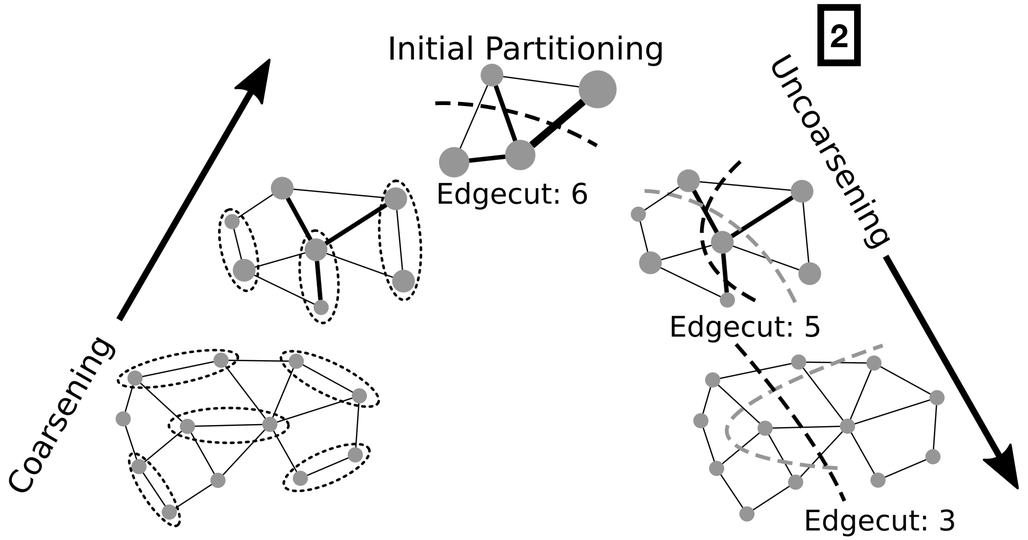
METIS

METIS stable version: 5.1.0, 3/30/2013
MT-METIS version: 0.7.3, 6/4/2020

- **METIS** - “is a ***set of serial programs for partitioning graphs***, partitioning finite element ***meshes***, and producing fill reducing orderings for ***sparse matrices***. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel *k*-way, and multi-constraint partitioning schemes developed in our lab.”

- **MT-METIS** - is a multi-threaded implementation of the **METIS** library





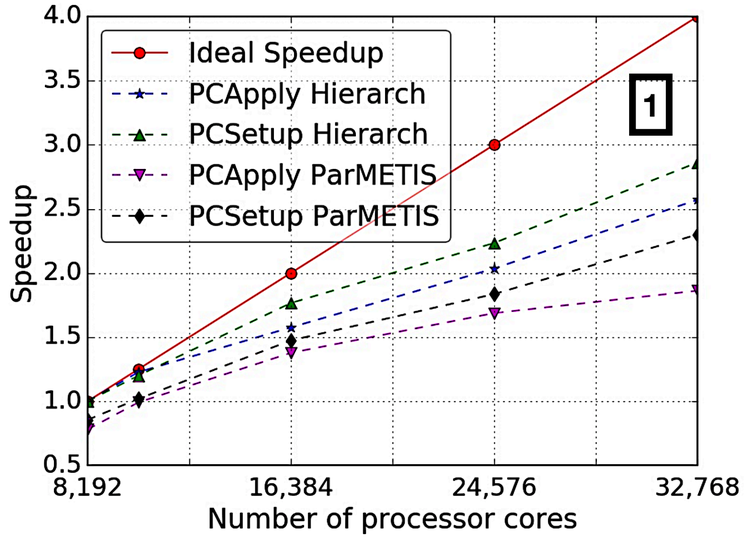
1. Gerasimos Arvanitis and Aris S. Lalos and Konstantinos Moustakas. (2018). Block-Based Spectral Processing of Static and Dynamic 3D Meshes using Orthogonal Iterations. eess.SP.
2. Dominique LaSalle and George Karypis. (2016). A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning. 45th International Conference on Parallel Processing (ICPP).

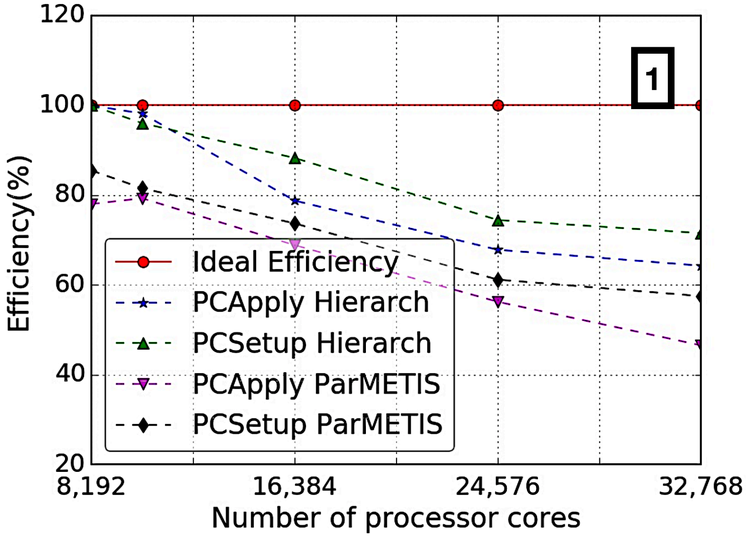
Domain decomposition

METIS

ParMETIS stable version: 4.0.3, 3/30/2013

- **ParMETIS** - “is an **MPI-based parallel library** that implements a variety of algorithms for partitioning **unstructured graphs**, **meshes**, and for computing fill-reducing orderings of **sparse matrices**. The algorithms implemented in ParMETIS are based on the parallel multilevel *k*-way graph-partitioning, adaptive repartitioning, and parallel multi-constrained partitioning schemes developed in our lab.”





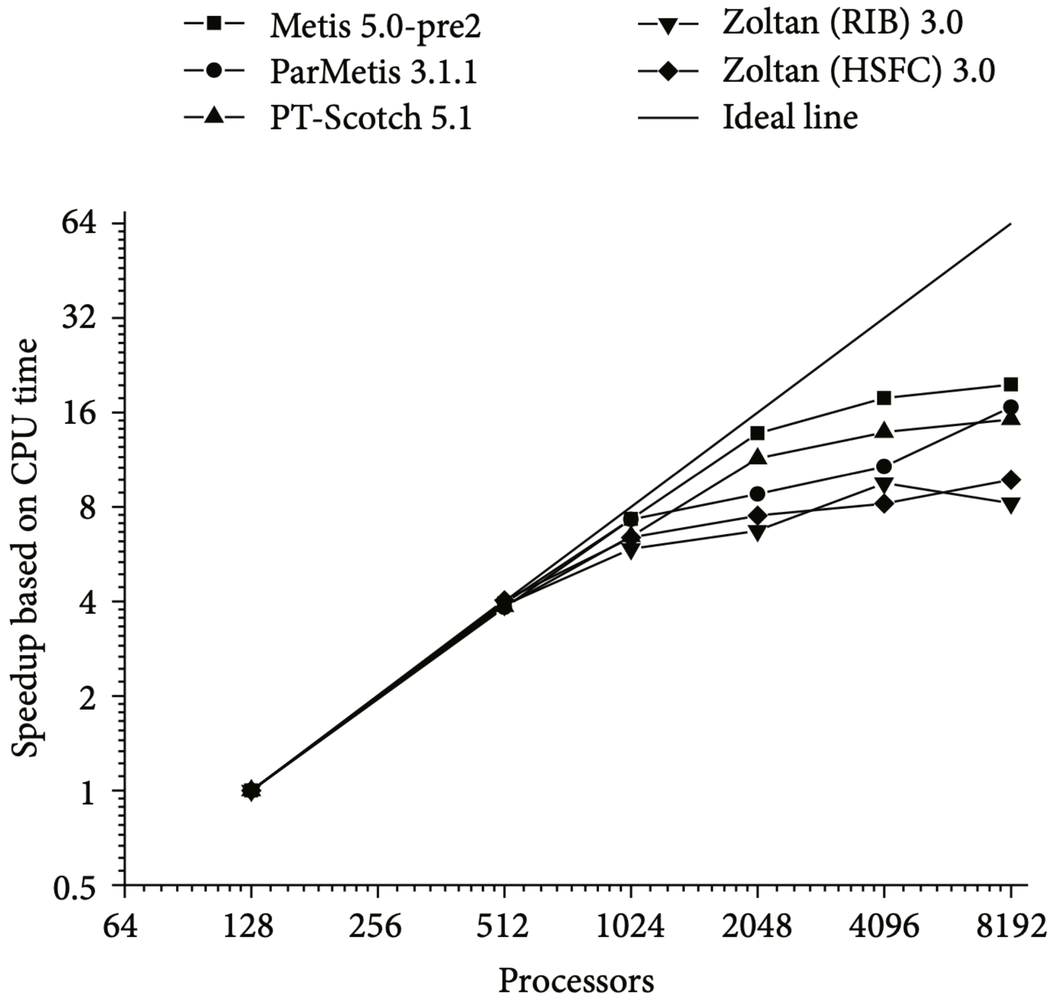
1. F. Kong *et al.*, "A General-Purpose Hierarchical Mesh Partitioning Method with Node Balancing Strategies for Large-Scale Numerical Simulations," *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, 2018, pp. 65-72

Domain decomposition

Other Libraries

- The final partitioning affects the overall performance of the main application by:
  - **Interprocess area** - how many elements belong to the interprocess edges?
  - **Distribution of the sub-domains** - how optimal is the subdomain numbering?
  - **Balanced workload** - how optimal is the partitioning?

CFD simulations with **Code_Saturne**



1. Zhi Shang, “Large-Scale CFD Parallel Computing Dealing with Massive Mesh”, Journal of Engineering, Volume 2013, Article ID 850148

Domain decomposition

METIS-5.x

- Provides API and **stand-alone programs**: gpmetis, mpmetis, ndmetis, m2gmetis
- Works with files (**IO**): graphs and meshes
- Allows to decompose **graphs** and **meshes**
- Allows to **convert mesh to a graph**
- Allows to **use weights** during partitioning
- Documentation: **http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf**

```
/* Is used to partition a graph into k parts using
either multilevel recursive bisection or multilevel k-
way partitioning */
METIS_API(int) METIS_PartGraphRecursive(…);
METIS_API(int) METIS_PartGraphKway(…);

/* This function is used to partition a mesh into k
parts based on a partitioning of the mesh's nodal
graph. */
METIS_API(int) METIS_PartMeshNodal(…);

/* This function is used to partition a mesh into k
parts based on a partitioning of the mesh's dual graph.
*/
METIS_API(int) METIS_PartMeshDual(…);

/* This function computes fill reducing orderings of
sparse matrices using the multilevel nested dissection
algorithm */
METIS_API(int) METIS_NodeND(…);

/* This function is used to generate the dual graph of
a mesh. */
METIS_API(int) METIS_MeshToDual(…);

/* This function is used to generate the nodal graph of
a mesh. */
METIS_API(int) METIS_MeshToNodal(…);
```

Domain decomposition

METIS-5.x

```
METIS_API(int) METIS_PartGraphRecursive(idx_t *nvtxs,    // the number of vertices (nodes) in the graph
                                        idx_t *ncon,     // the number of balancing constraints. It should be
                                                         // at least 1
                                        idx_t *xadj,     // the adjacency structure of the graph (offsets)
                                        idx_t *adjncy,   // the adjacency structure of the graph (adjacency list)
                                        idx_t *vwgt,     // the weights of the vertices (nodes) (NULL)
                                        idx_t *vsize,    // the size of the vertices for computing the total
                                                         // communication volume (NULL)
                                        idx_t *adjwgt,   // the weights of the edges (NULL)
                                        idx_t *nparts,   // the number of parts to partition the graph
                                        real_t *tpwgts,  // an array of size nparts×ncon that specifies the
                                                         // desired weight for each partition and constraint (NULL)
                                        real_t *ubvec,   // an array of size ncon that specifies the allowed load
                                                         // imbalance tolerance for each constraint (NULL)
                                        idx_t *options,  // an array of options (NULL)
                                        idx_t *edgecut,  // upon successful completion, this variable stores the
                                                         // edge-cut or the total communication volume of the
                                                         // partitioning solution
                                        idx_t *part);    // a vector of size nvtxs that upon successful completion
                                                         // stores the partition vector of the graph
```

Domain decomposition

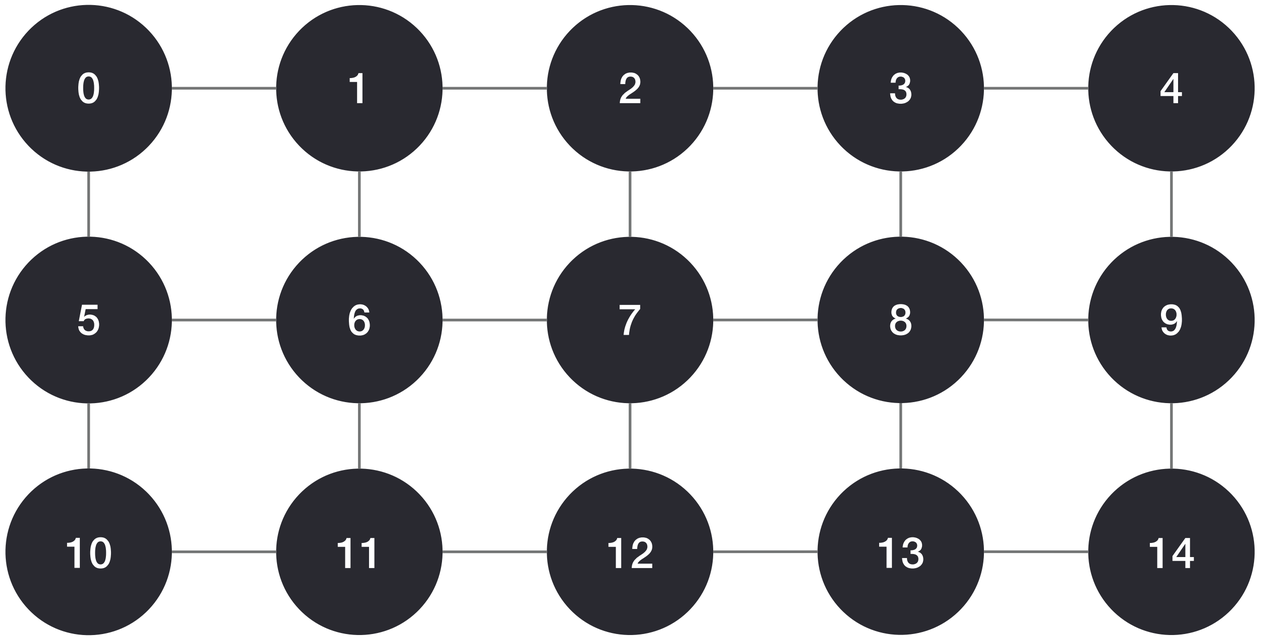
METIS-5.x

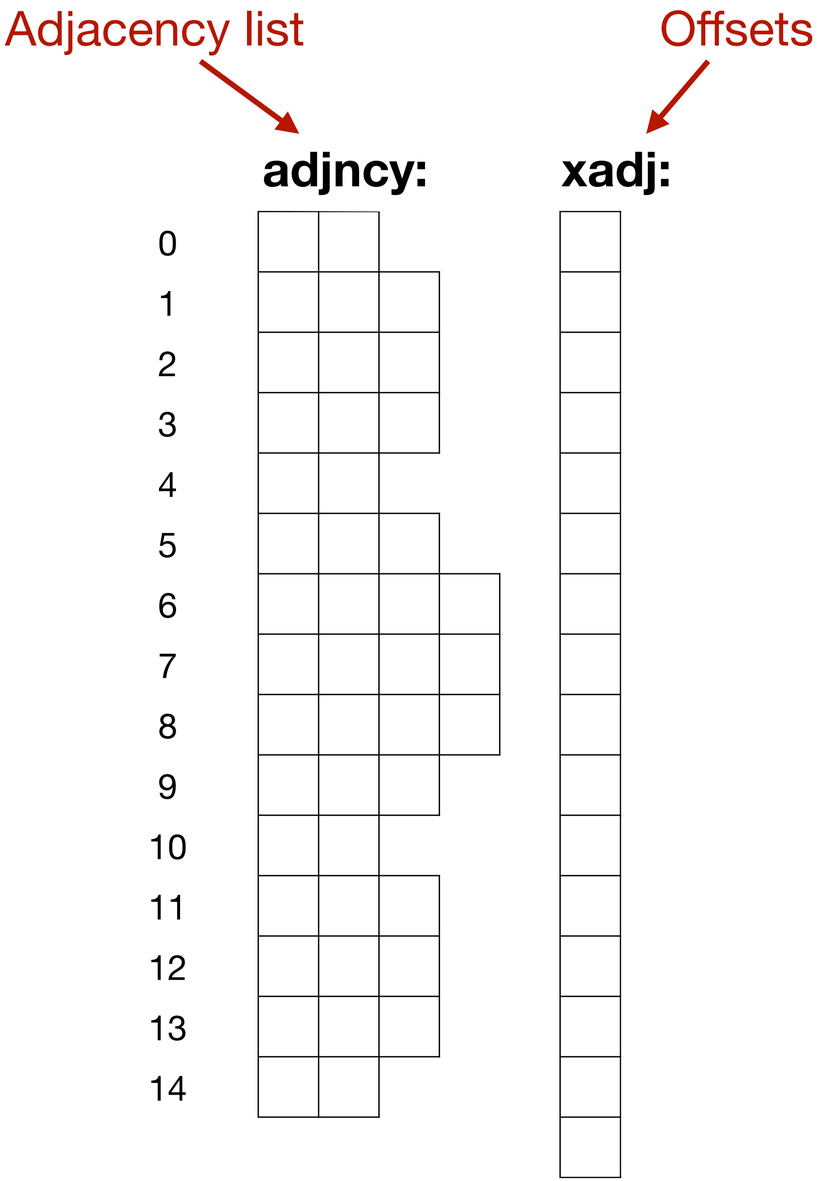
```
METIS_API(int) METIS_PartGraphKway(idx_t *nvtxs,   // the number of vertices (nodes) in the graph
                                   idx_t *ncon,    // the number of balancing constraints. It should be
                                                   // at least 1
                                   idx_t *xadj,    // the adjacency structure of the graph (offsets)
                                   idx_t *adjncy,  // the adjacency structure of the graph (adjacency list)
                                   idx_t *vwgt,    // the weights of the vertices (nodes) (NULL)
                                   idx_t *vsize,   // the size of the vertices for computing the total
                                                   // communication volume (NULL)
                                   idx_t *adjwgt,  // the weights of the edges (NULL)
                                   idx_t *nparts,  // the number of parts to partition the graph
                                   real_t *tpwgts, // an array of size nparts×ncon that specifies the
                                                   // desired weight for each partition and constraint (NULL)
                                   real_t *ubvec,  // an array of size ncon that specifies the allowed load
                                                   // imbalance tolerance for each constraint (NULL)
                                   idx_t *options, // an array of options (NULL)
                                   idx_t *edgecut, // upon successful completion, this variable stores the
                                                   // edge-cut or the total communication volume of the
                                                   // partitioning solution
                                   idx_t *part);   // a vector of size nvtxs that upon successful completion
                                                   // stores the partition vector of the graph
```

Domain decomposition

METIS-5.x

What are **xadj** and **adjncy** exactly?

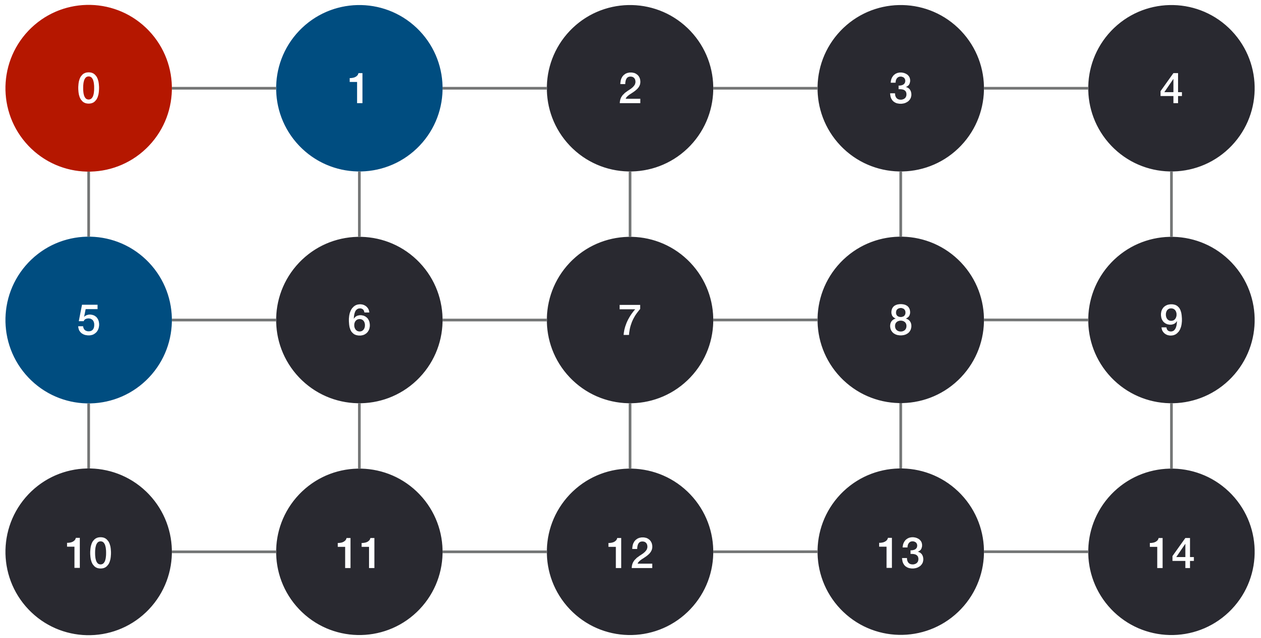


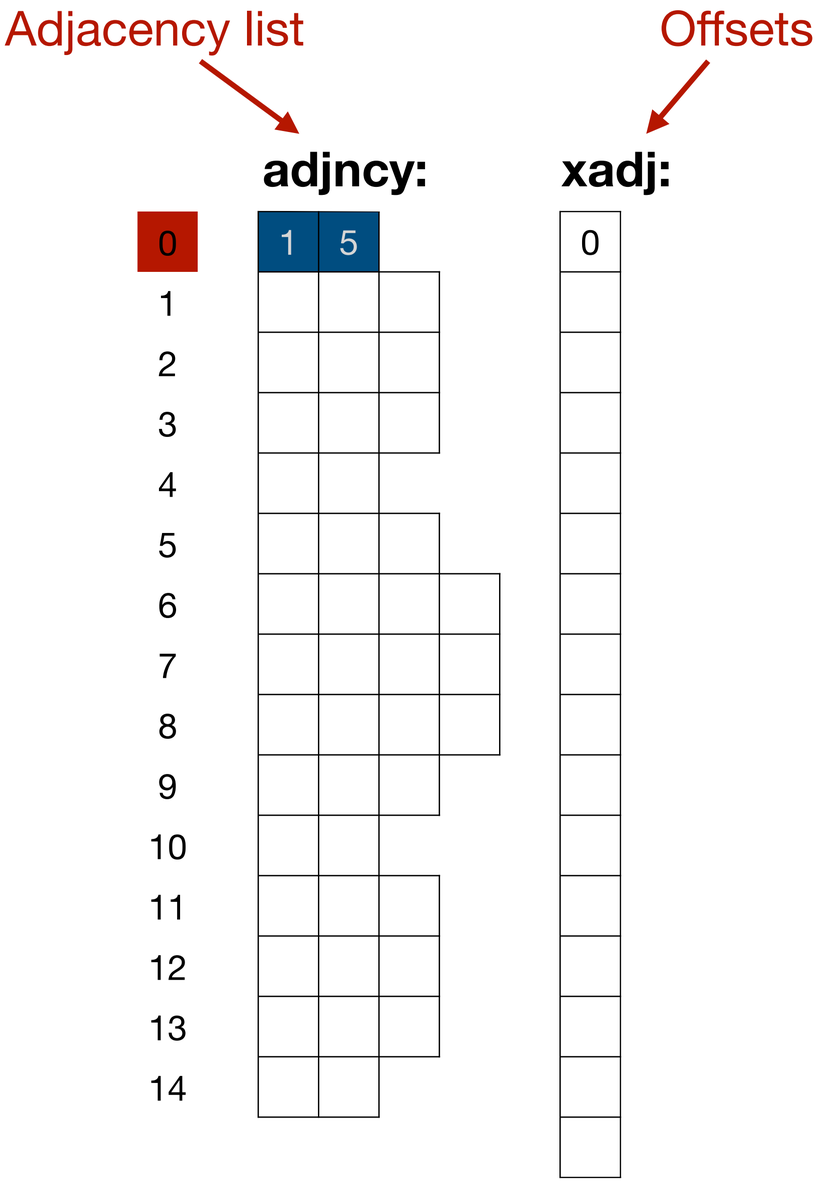


Domain decomposition

METIS-5.x

What are **xadj** and **adjncy** exactly?

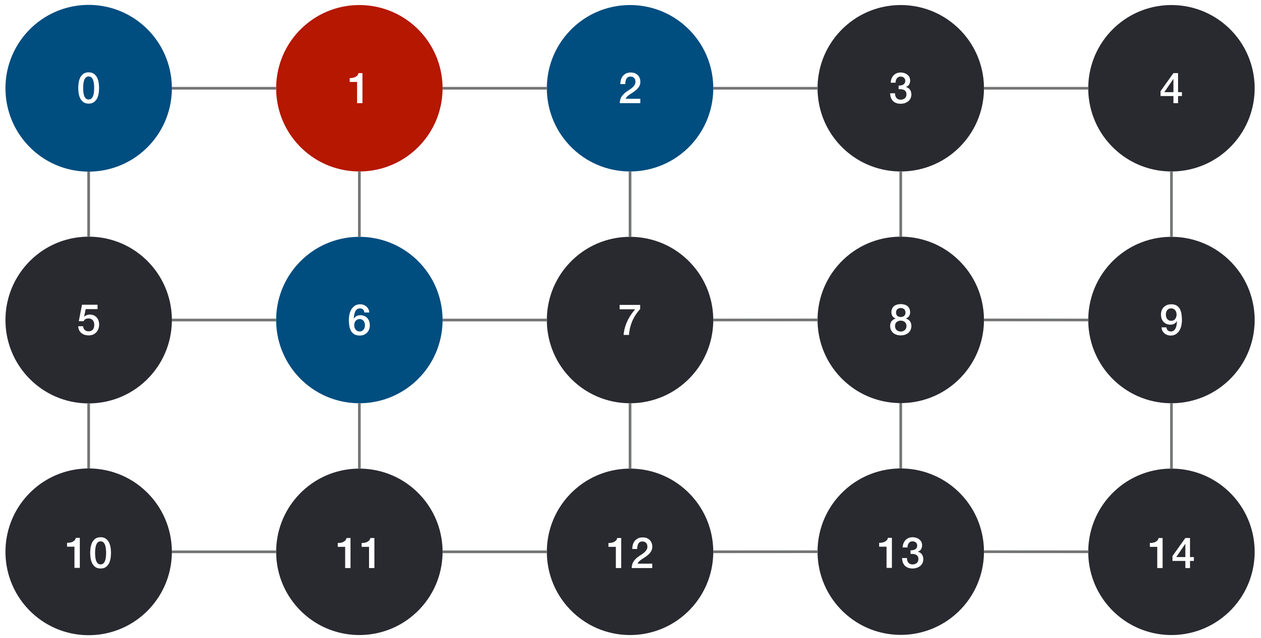


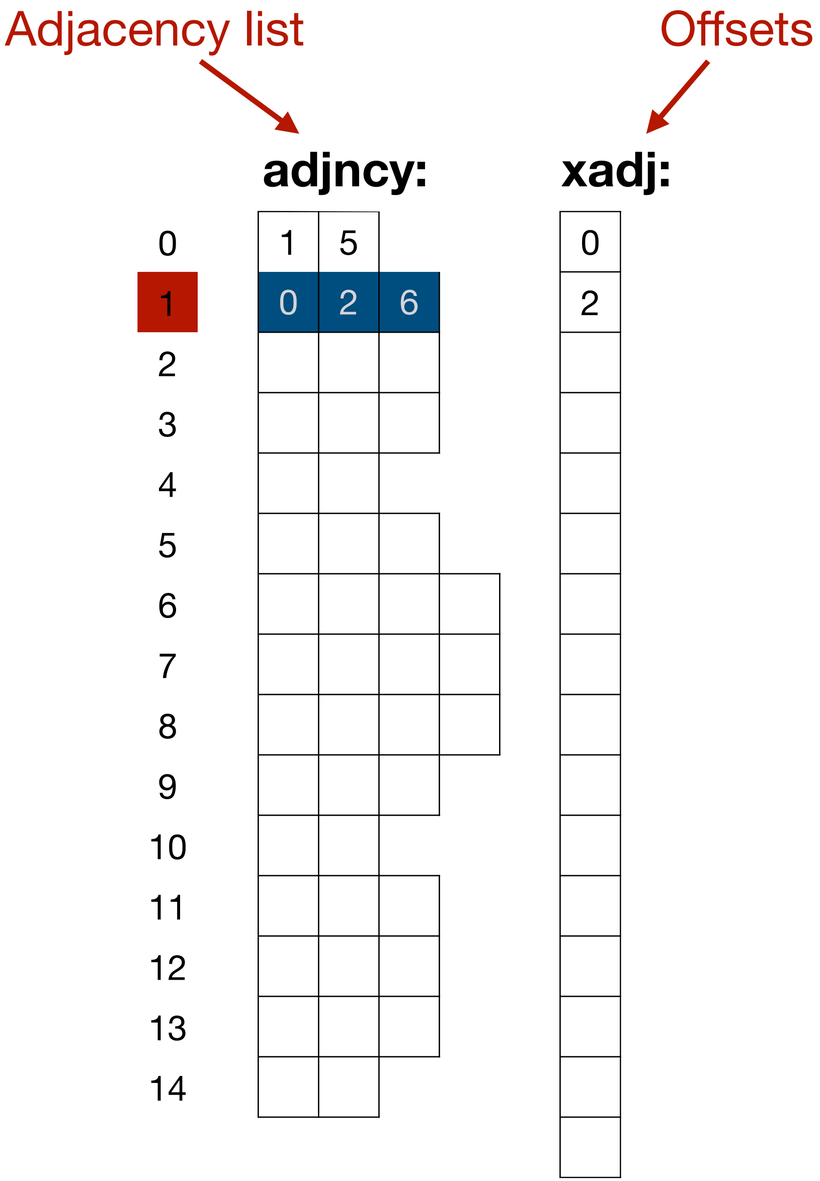


Domain decomposition

METIS-5.x

What are **xadj** and **adjncy** exactly?

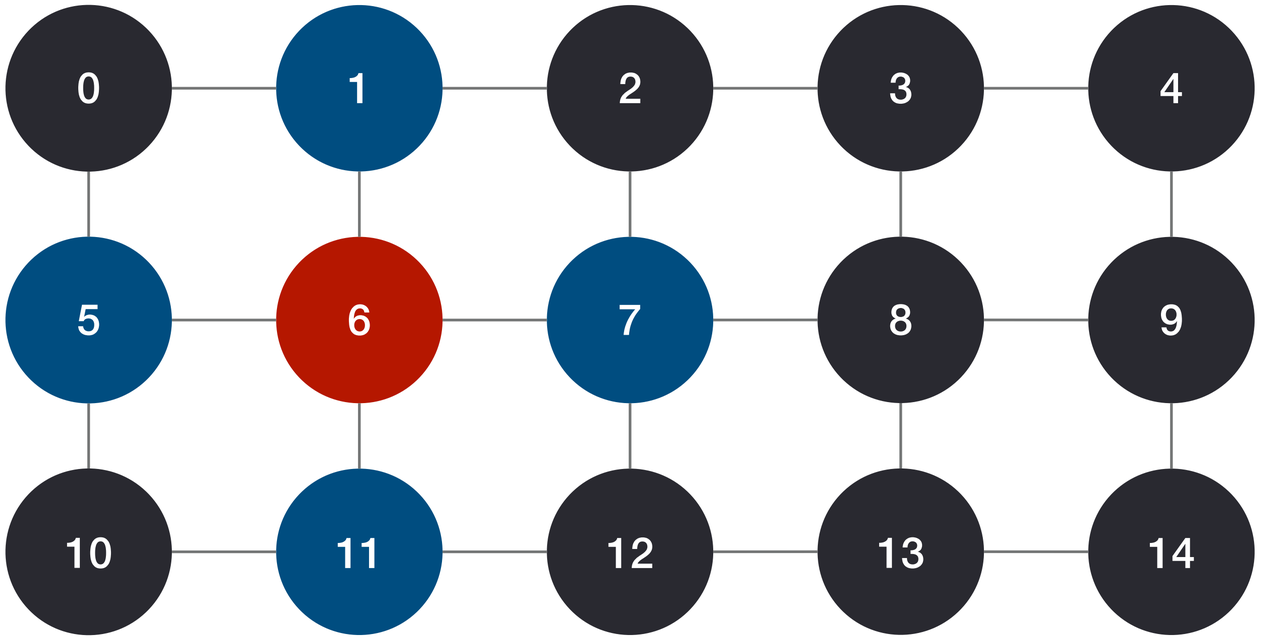


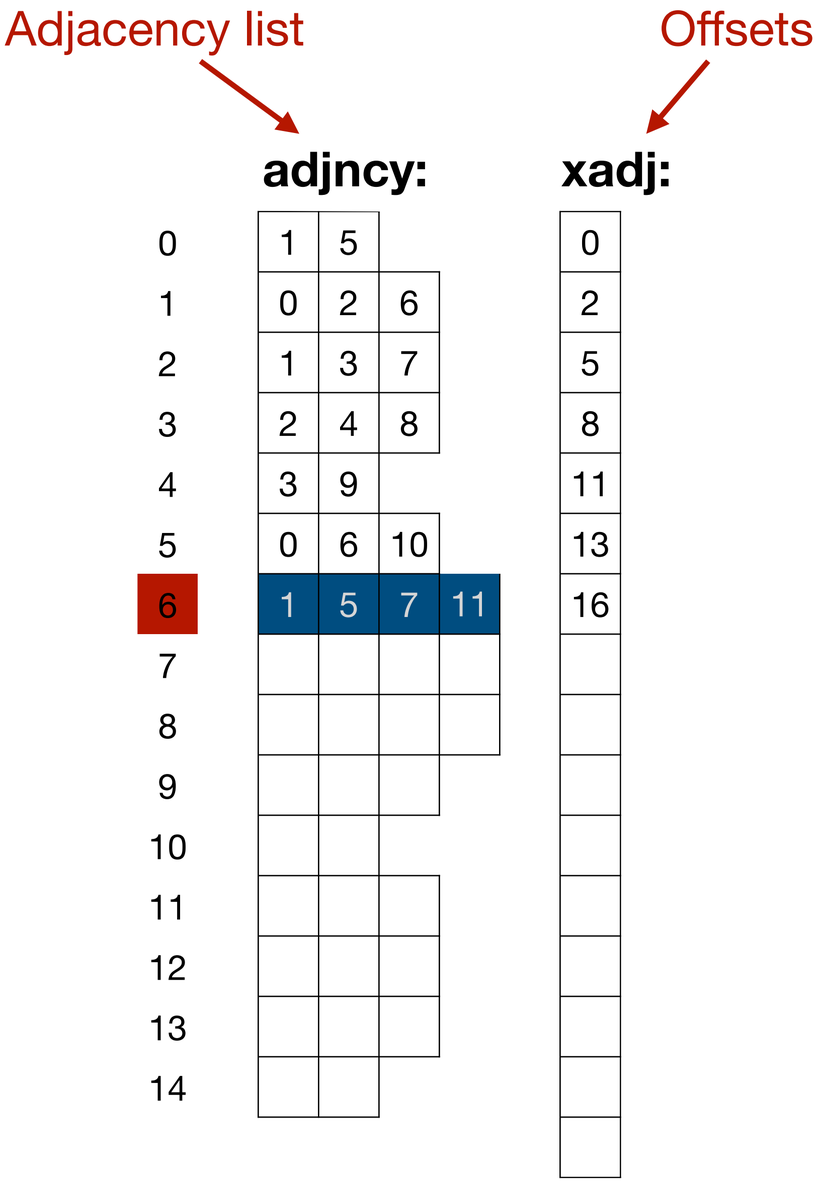


Domain decomposition

METIS-5.x

What are **xadj** and **adjncy** exactly?

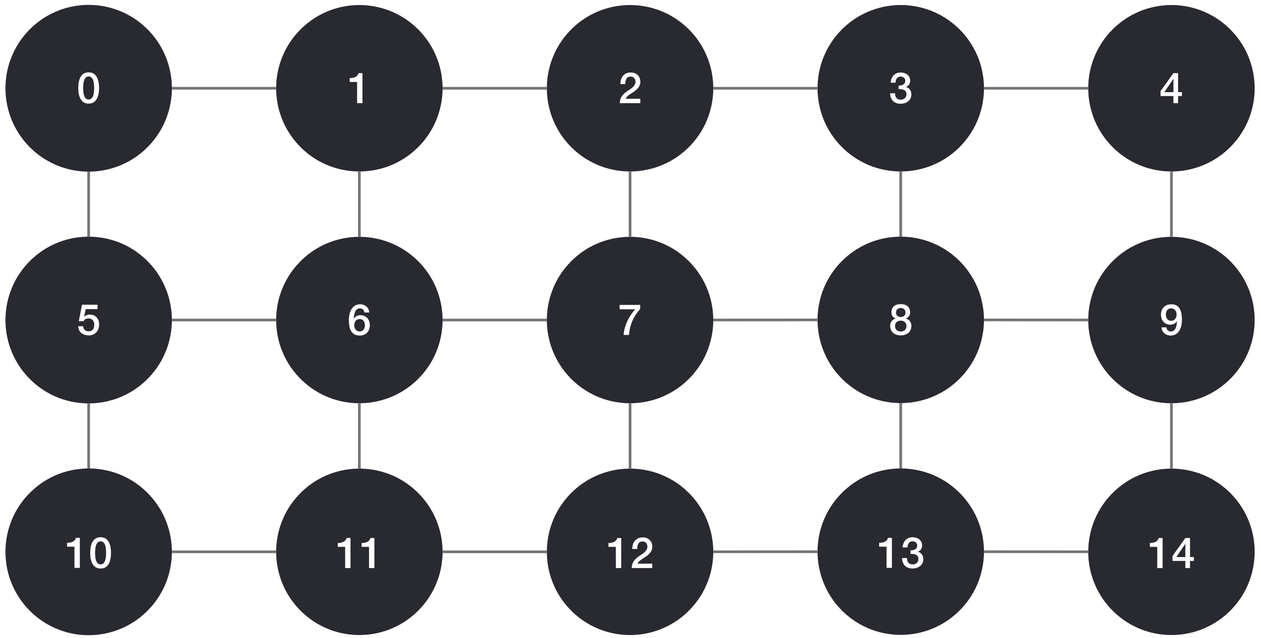




Domain decomposition

METIS-5.x

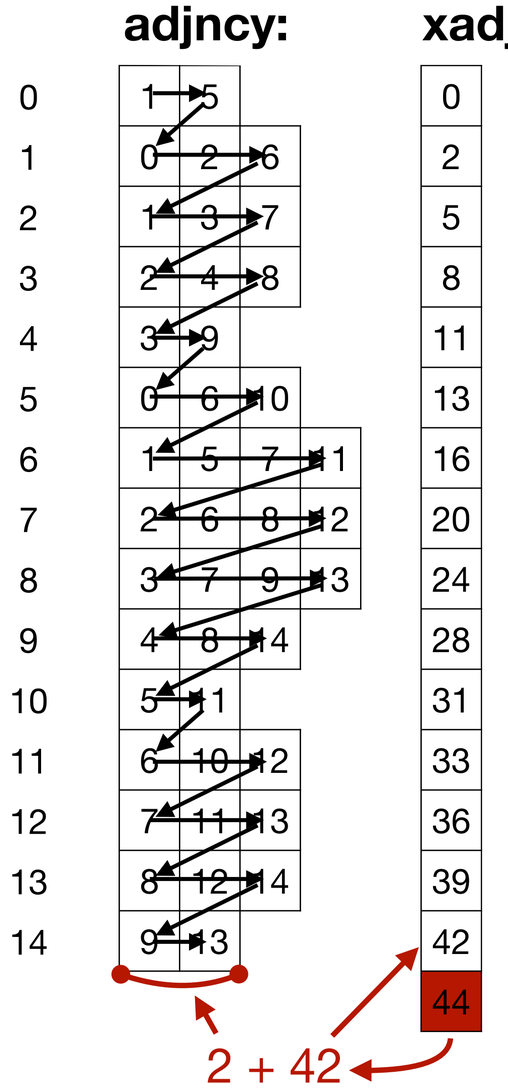
What are **xadj** and **adjncy** exactly?



This is actually a 1D contiguous array!

Adjacency list

Offsets

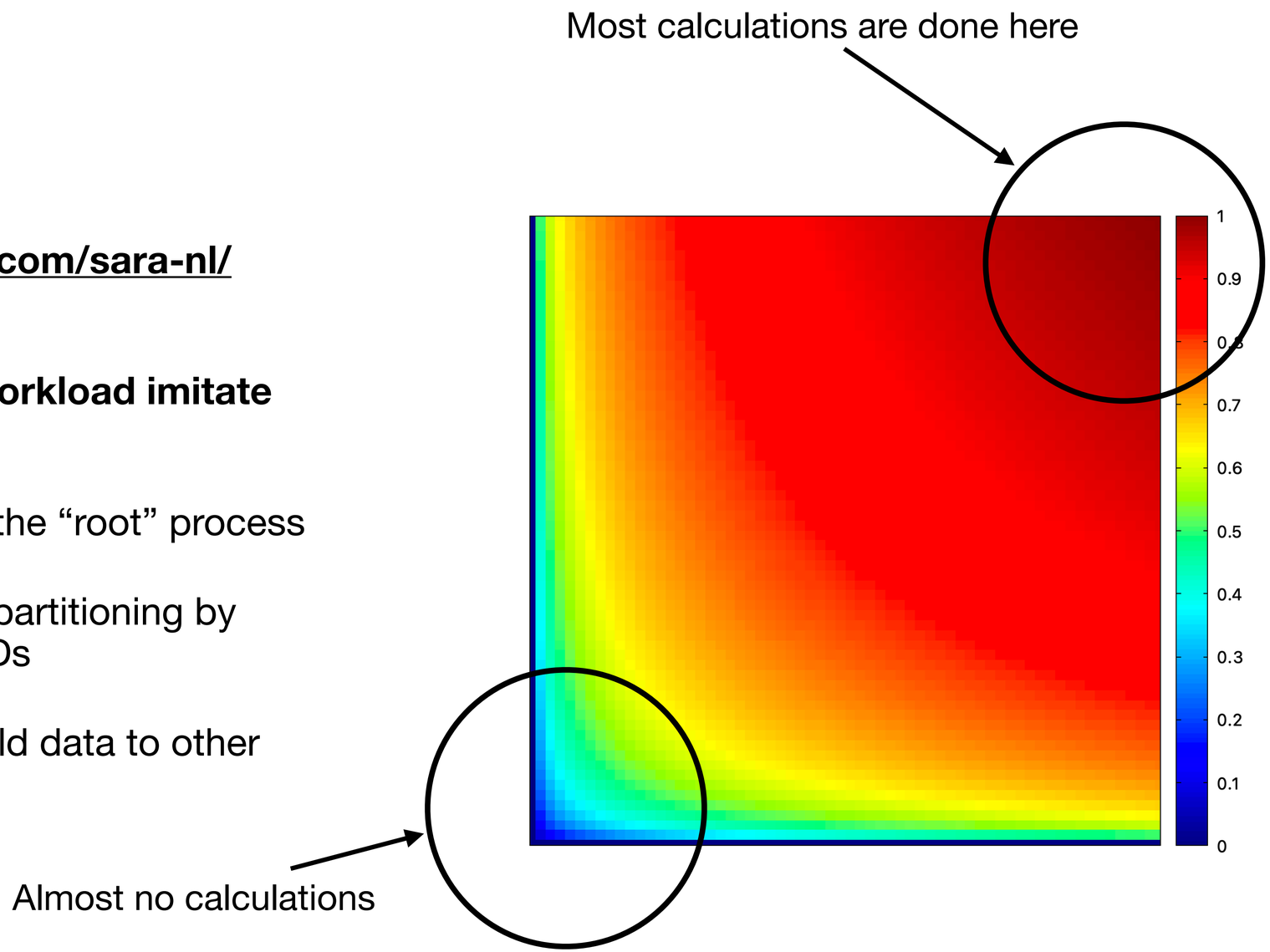


Hands-on #2.3

Hands-on

Decomposition

- Clone the code from **https://github.com/sara-nl/prace_topologies.git**
- **Using a dimensionless field of a workload imitate an uneven load balance**
  - The field is assembled locally by the “root” process
  - The “root” process performs the partitioning by filling in an auxiliary array with PIDs
  - The array is later used to send field data to other processes

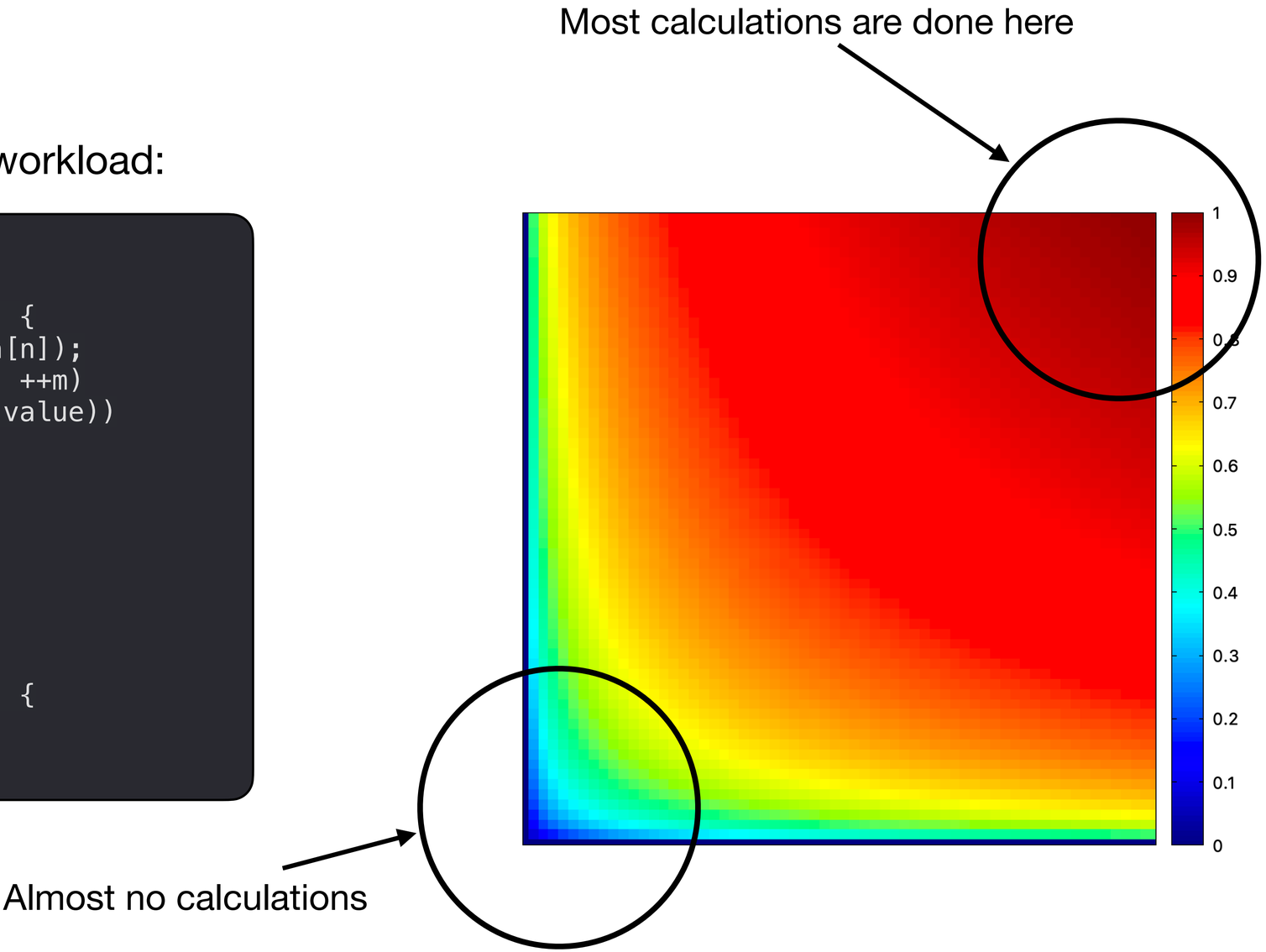


Hands-on

Decomposition

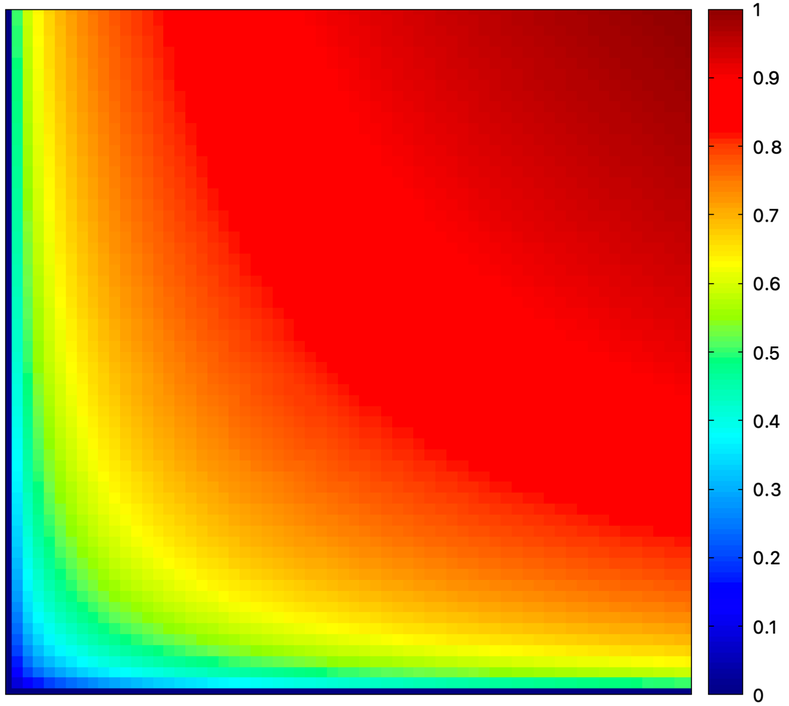
- Example of the imitated “dummy” workload:

```
double Field::performDummyWork() {
    double result = 0.;
    for(int n = 0; n < data.size(); ++n) {
        double value = getLocalLoad(data[n]);
        for (int m = 0; m < (int) value; ++m)
            result += (log(value) + cos(value))
                    / exp(value);
        }
    }
    return result;
}

…

inline double getLocalLoad(double value) {
    return exp(15. * value);
}
```

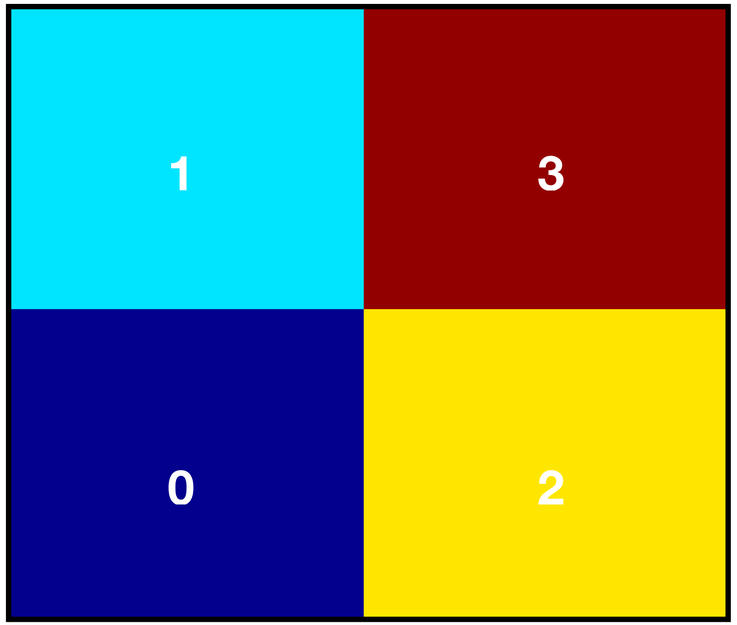


Hands-on

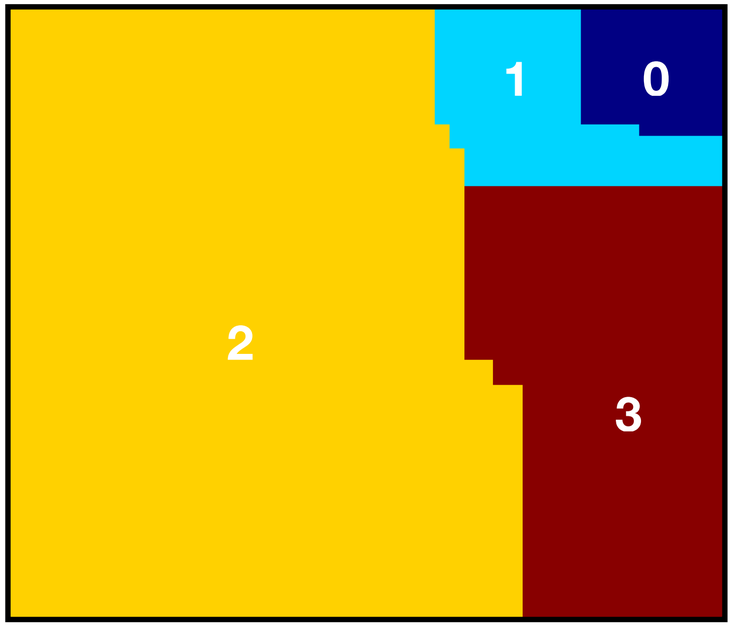
Decomposition



Structured decomposition



Graph-based decomposition with weights



Hands-on

Decomposition

- Use **METIS_PartGraphKway()** to implement graph partitioning with METIS:
  - ***src/MPI/Decomposition/decompositionMETIS.cpp:decompose()***
  - Set **vsize**, **adjwgt**, **tpwgts**, **ubvec**, options to *NULL*
- Compile the code with OpenMPI and test the performance
  - change the execution type between **"*-t s*"** (*STRUCTURED*) and "***-t m***" (*METIS*)
- In ***main.cpp***, look for the vector *weights* and change its values. See what happens to the distribution and performance
- Replace the *weights* argument with *NULL* in **METIS_PartGraphKway()**
- Visualise the final decomposition using *gnuplot*
- Consider different problem sizes and different number of processes (e.g. a grid of 10x10, 32x32, 64x64 cells)

Hands-on

Decomposition

```
$ module load 2022
$ module load foss/2022a
$ module load METIS/5.1.0-GCCcore-11.3.0
$ module load gnuplot/5.4.4-GCCcore-11.3.0
$
$ git clone https://github.com/sara-nl/prace_topologies.git
$ cd prace_topologies
$ ./make_all.sh
$
$ mpirun -n 4 ./topologies -s 10 10 -d 2 2 -t m
...
$ mpirun -n 4 ./topologies -s 10 10 -d 2 2 -t s
...
$
$ # if connected with X11 forwarding
$ gnuplot
gnuplot> p “graph.dat” matrix w ima
gnuplot> p “original_0.dat” matrix w ima
```

```
Use the following keys:
  -s - set number of the grid cells in each direction (i j)
  -d - set decomposition for each direction (i j)
       (doesn’t affect the METIS decomposition, but should be
        set anyway!)
  -t - set decomposition type ('m' for METIS, 's' for STRUCTURED)
```

MPI Topologies

MPI Topologies

Communicators

- A Communicator can represent a topology of the decomposed domain (or not - **MPI_COMM_WORLD**)

Cartesian topology

Pros:

- *2D/3D structured grids*
- *Allows access to neighbours via indices*
- *Provides “cyclic” boundaries*

Cons:

- *Domain should be decomposed in a structured way*
- *Can’t connect diagonal neighbours*

Graph topology

Pros:

- *Works for any type of grids*
- *Works with arbitrary neighbourhood*
- *Provides a way to look up for a neighbourhood or any process from any process*

Cons:

- *Stores an instance of the graph at every process (non-scalable)*
- ***Legacy (will be removed from the next standard)***

Distributed (adjacent) graph topology

Pros:

- *Works for any type of grids*
- *Works with an arbitrary neighbourhood*
- *Fully distributed (less memory)*
- *Scalable*

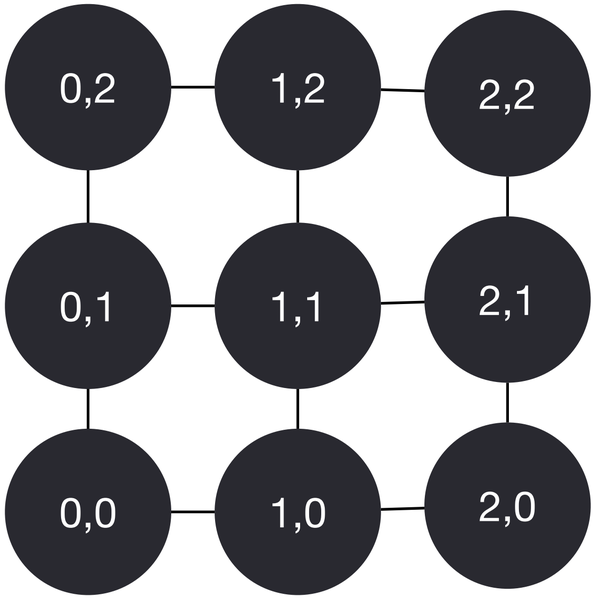
Cons:

- *It is a graph :)*

MPI Topologies

Cartesian

- The basic topology for structured grids
- The **reorder** info is ignored by some implementations
- Access to the neighbouring processes is done using a “shift” operation




```
int MPI_Cart_create(MPI_Comm old_comm,    // input communicator
                    int ndims,            // number of dimensions of cartesian grid
                    const int dims[],     // integer array of size ndims specifying the number of processes
                                          // in each dimension
                    const int periods[],  // logical array of size ndims specifying whether the grid is periodic
                                          // (true) or not (false) in each dimension
                    int reorder,          // ranking may be reordered (true) or not (false)
                    MPI_Comm *comm_cart); // communicator with new cartesian topology
```

MPI Topologies

Cartesian

```
/* Determines process rank in communicator given Cartesian location */
int MPI_Cart_rank(MPI_Comm comm,         // communicator with cartesian structure
                  const int coords[],   // integer array (of size ndims, the number of dimensions of the Cartesian
                                        // topology associated with comm) specifying the cartesian coordinates of
                                        // a process
                  int *rank);           // rank of specified process

/* Determines process coords in cartesian topology given rank in group */
int MPI_Cart_coords(MPI_Comm comm,       // communicator with cartesian structure
                    int rank,           // rank of a process within group of comm
                    int maxdims,        // length of vector coords in the calling program
                    int coords[]);      // integer array (of size ndims) containing the Cartesian coordinates of
                                        // specified process

/* Returns the shifted source and destination ranks, given a shift direction and amount */
int MPI_Cart_shift(MPI_Comm comm,        // communicator with cartesian structure
                   int direction,       // coordinate dimension of shift
                   int disp,            // displacement (> 0: upwards shift, < 0: downwards shift)
                   int *rank_source,    // rank of source process
                   int *rank_dest);     // rank of destination process
```

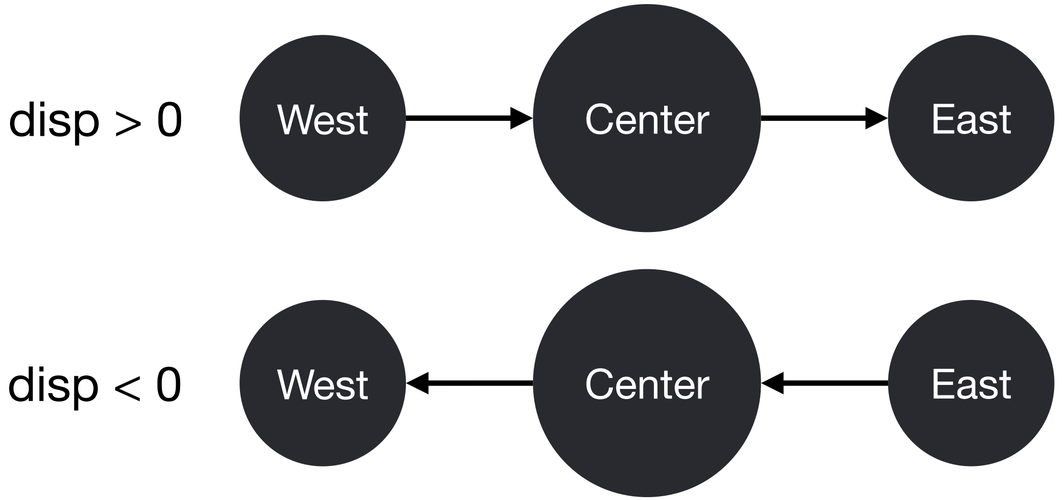
Simple and clear

What is source/destination process?

MPI Topologies

Cartesian

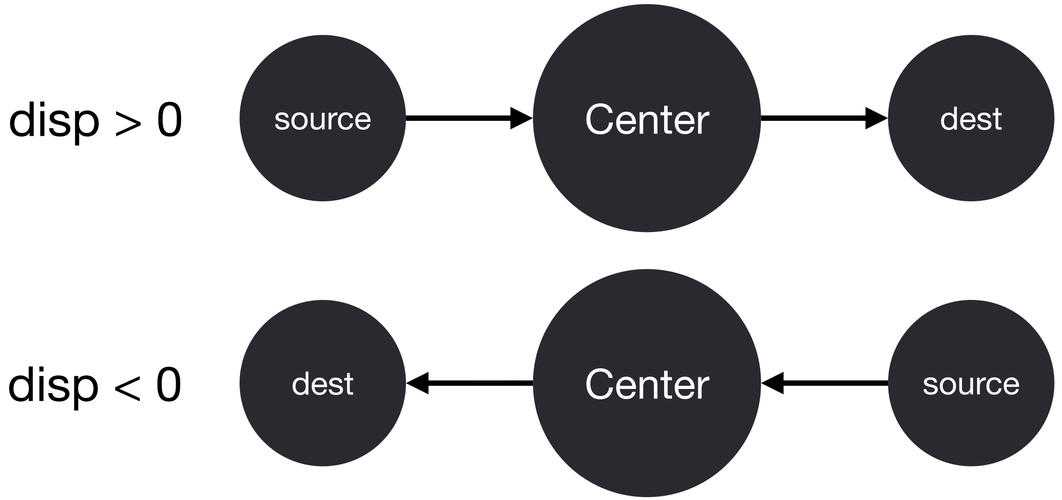
```
/* Returns the shifted source and destination ranks, given a shift direction and amount */
int MPI_Cart_shift(MPI_Comm comm,       // communicator with cartesian structure
                   int direction,       // coordinate dimension of shift
                   int disp,            // displacement (> 0: upwards shift, < 0: downwards shift)
                   int *rank_source,    // rank of source process
                   int *rank_dest);     // rank of destination process
```



MPI Topologies

Cartesian

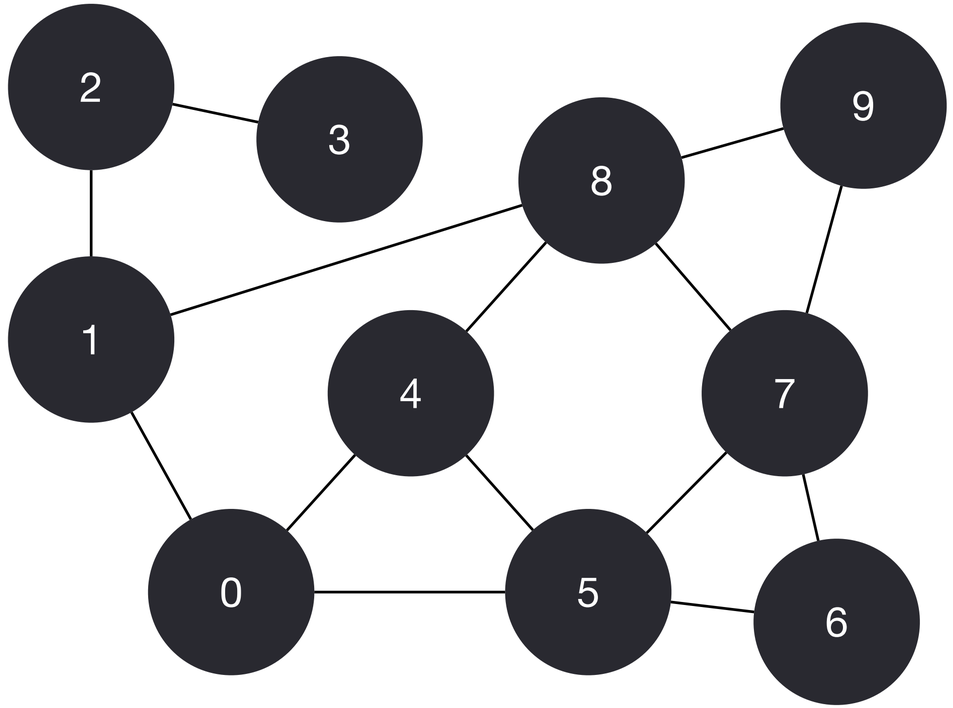
```
/* Returns the shifted source and destination ranks, given a shift direction and amount */
int MPI_Cart_shift(MPI_Comm comm,       // communicator with cartesian structure
                   int direction,       // coordinate dimension of shift
                   int disp,            // displacement (> 0: upwards shift, < 0: downwards shift)
                   int *rank_source,    // rank of source process
                   int *rank_dest);     // rank of destination process
```



If the source or destination process does not exist, then the corresponding rank is set to **MPI_PROC_NULL**

MPI Topologies

Distributed graph

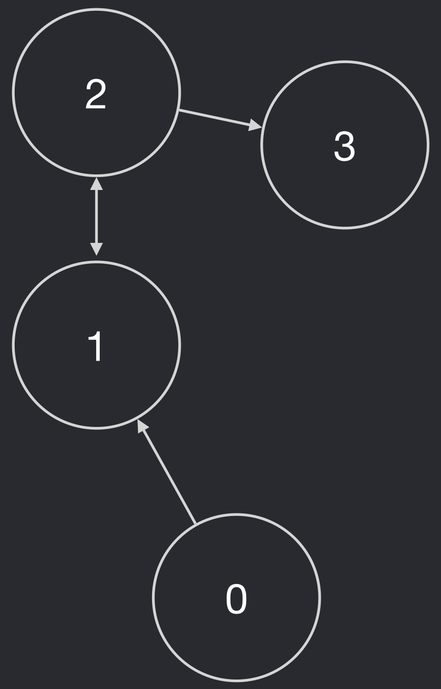


- The **reorder** info is ignored by some implementations
- Access to the neighbouring processes is done using the adjacency information

```
int MPI_Dist_graph_create(MPI_Comm comm_old,   // input communicator
                          int n,               // number of source nodes for which this process specifies edges
                                               // (non-negative integer)
                          const int nodes[],   // array containing the n source nodes for which this process
                                               // specifies edges (array of non-negative integers)
                          const int degrees[], // array specifying the number of destinations for each source node
                                               // in the source node array (array of non-negative integers)
                          const int targets[], // destination nodes for the source nodes in the source node array
                                               // (array of non-negative integers)
                          const int weights[], // weights for source to destination edges (array of non-negative
                                               // integers or MPI_UNWEIGHTED)
                          MPI_Info info,       // hints on optimization and interpretation of weights
                          int reorder,         // the process may be reordered (true) or not (false)
                          MPI_Comm * newcomm); // communicator with distributed graph topology added
```

MPI Topologies

Distributed graph




```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,          // input communicator
                                   int indegree,               // size of sources and sourceweights arrays
                                                               // (non-negative integer)
                                   const int sources[],        // ranks of processes for which the calling process
                                                               // is a destination (array of non-negative
                                                               // integers)
                                   const int sourceweights[],  // weights of the edges into the calling process
                                                               // (array of non-negative integers or
                                                               // MPI_UNWEIGHTED)
                                   int outdegree,              // size of destinations and destweights arrays
                                                               // (non-negative integer)
                                   const int destinations[],   // ranks of processes for which the calling process
                                                               // is a source (array of non-negative integers)
                                   const int destweights[],    // weights of the edges out of the calling process
                                                               // (array of non-negative integers or
                                                               // MPI_UNWEIGHTED)
                                   MPI_Info info,              // hints on optimization and interpretation of
                                                               // weights (handle)
                                   int reorder,                // the process may be reordered (true) or not
                                                               // (false)
                                   MPI_Comm *comm_dist_graph); // communicator with distributed graph topology
                                                               // added
```

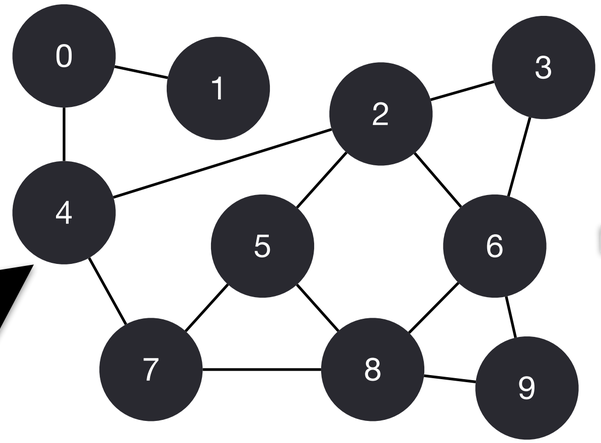
MPI Topologies

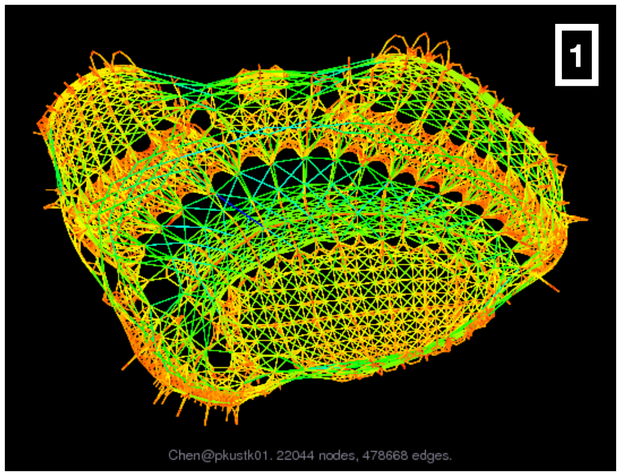
Distributed graph

```
/* Provides the neighbours of the calling process in a distributed graph topology */
int MPI_Dist_graph_neighbors(MPI_Comm comm,           // communicator with distributed graph topology
                             int maxindegree,         // size of sources and sourceweights arrays (non-negative
                                                      // integer)
                             int sources[],           // processes for which the calling process is a destination
                                                      // (array of non-negative integers)
                             int sourceweights[],     // weights of the edges into the calling process (array of
                                                      // non-negative integers)
                             int maxoutdegree,        // size of destinations and destweights arrays
                                                      // (non-negative integer)
                             int destinations[],      // processes for which the calling process is a source
                                                      // (array of non-negative integers)
                             int destweights[]);      // weights of the edges out of the calling process (array
                                                      // of non-negative integers)

/* Provides the number of edges into and out of the process */
int MPI_Dist_graph_neighbors_count(MPI_Comm comm,     // communicator with distributed graph topology
                                   int *inneighbors,  // number of edges into this process
                                   int *outneighbors, // number of edges out of this process
                                   int *weighted);    // false if MPI_UNWEIGHTED was supplied during creation,
                                                      // true otherwise
```

MPI Topologies

Reordering

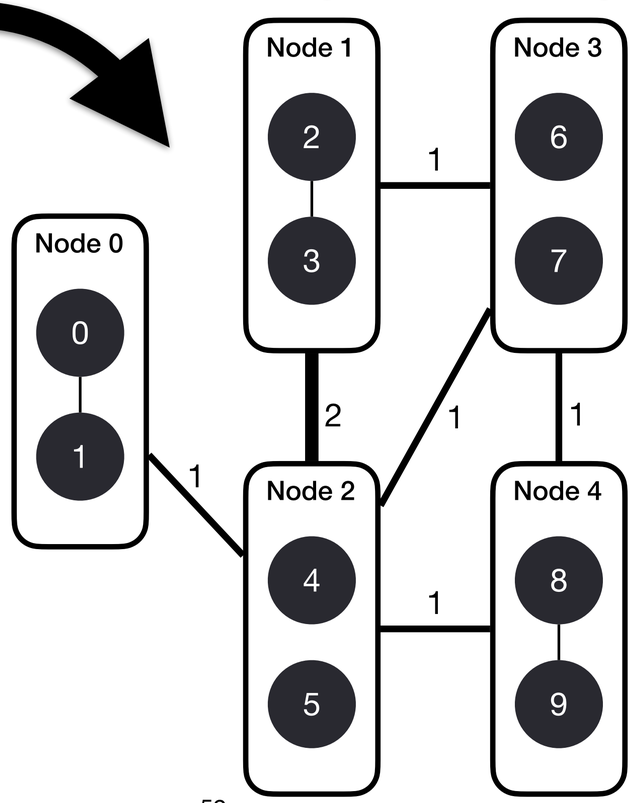




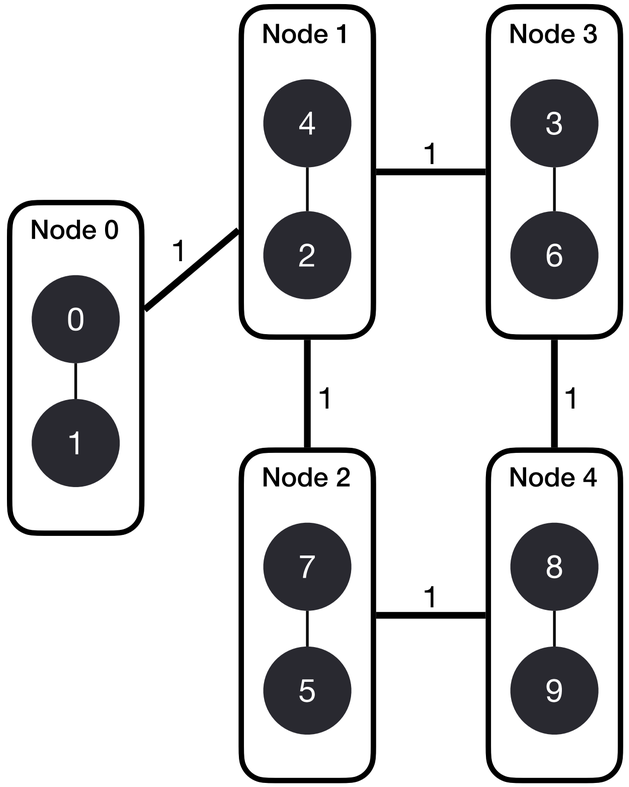
1. https://www.cise.ufl.edu/research/sparse/matrices/Chen/pkustk01.html

Reordering allowed to reduce the number of inter-node communications from 7 to 5

Original mapping



Optimised mapping



MPI Topologies

Operations

- Using a communicator built for a specific topology, it is possible to invoke “local" (neighbourhood) collective communications
- It’s also possible to create a local sub-communicator and use other collective communications

```
int MPI_Neighbor_allgather(…);
int MPI_Ineighbor_allgather(…);
int MPI_Neighbor_allgatherv(…);
int MPI_Ineighbor_allgatherv(…);
int MPI_Neighbor_alltoall(…);
```

```
int MPI_Ineighbor_alltoall(…);
int MPI_Neighbor_alltoallv(…);
int MPI_Ineighbor_alltoallv(…);
int MPI_Neighbor_alltoallw(…);
int MPI_Ineighbor_alltoallw(…);
```

MPI Topologies

Topology Inquiry

- The topology information can be looked up using **MPI_Topo_test()**
- This can be used at runtime, for example, to define the logic flow of the code

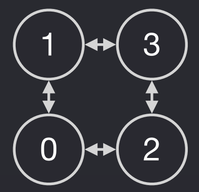
```
#define MPI_UNDEFINED          -32766     /* undefined stuff */
#define MPI_DIST_GRAPH         3          /* dist graph topology */
#define MPI_CART               1          /* cartesian topology */
#define MPI_GRAPH              2          /* graph topology */


/* Determines the type of topology (if any) associated with a communicator */
int MPI_Topo_test(MPI_Comm comm,          // communicator
                  int *status);           // topology type of communicator comm (integer). If the communicator
                                          // has  no associated topology, returns MPI_UNDEFINED
```

Hands-on #2.4

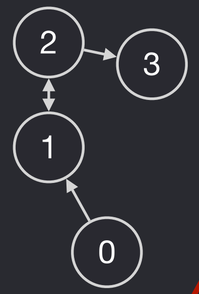
Hands-on

MPI Topologies

- Implement Cartesian and adjacent graph topologies:
  - ***src/MPI/topologies.cpp:createCartTopology()***
  - ***src/MPI/topologies.cpp:createGraphTopology()***
- For the adjacent graph topology use **MPI_UNWEIGHTED** and **MPI_INFO_NULL**
- In the *main* function, comment out the ***performDummyWork()*** calculations and add a call for:
  - ***createCartTopology()*** or
  - ***createGraphTopology()***




```
int MPI_Cart_create(MPI_Comm old_comm,
                    int ndims,
                    const int dims[],
                    const int periods[],
                    int reorder,
                    MPI_Comm *comm_cart);
```




```
int MPI_Dist_graph_create_adjacent(
                    MPI_Comm comm_old,
                    int indegree,
                    const int sources[],
                    const int sourceweights[],
                    int outdegree,
                    const int destinations[],
                    const int destweights[],
                    MPI_Info info,
                    int reorder,
                    MPI_Comm *comm_dist_graph);
```

Keep it bidirectional

Hands-on

MPI Topologies

- Test both topologies by implementing a few of the discussed functions.
  - Modify:
    - ***MPI/topologies.cpp:testCartTopology()***
    - ***MPI/topologies.cpp:testGraphTopology()***
  - Add calls for the corresponding functions to *main.cpp*
- Try the following:
  - Get process ID by coordinates and vice versa (Cartesian)
  - Get a list of the neighbouring processes (graph)
- Implement one of the neighbourhood collective communications (e.g. gather ranks)

```
/* Get rank from coordinates */
int MPI_Cart_rank();

/* Get coordinates from rank */
int MPI_Cart_coords();

/* Get neighbouring PIDs */
int MPI_Cart_shift();


/* Get number of neighbours */
int MPI_Dist_graph_neighbors_count();

/* Get a list of neighbours */
int MPI_Dist_graph_neighbors();

/* Gather from neighbours */
int MPI_Neighbor_allgather(…);
```