High-**P**erformance **C**omputing 20**24**

Distributed Memory Parallelism and MPI

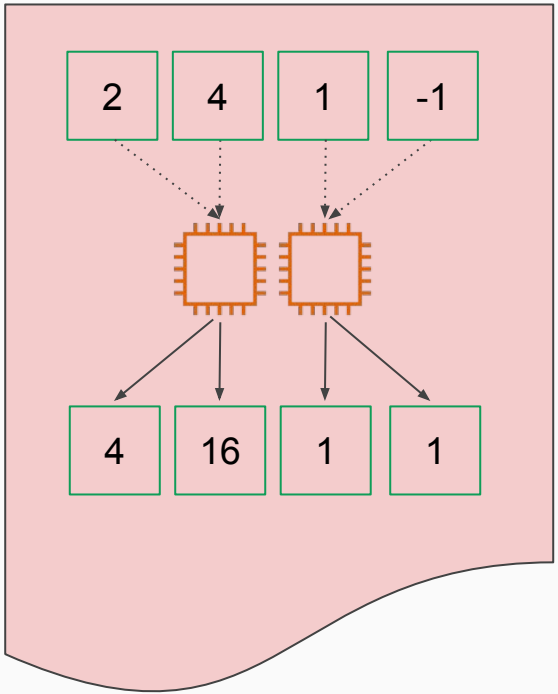
Aryan Eftekhari, Olaf Schenk | **U**niversità della **S**vizzera **i**taliana (USI) | **I**nstitute of **C**omputing (CI)

Review
OpenMP

Shared Memory Parallelization

*Computation is distributed along **threads**.*

Synchronization between threads.



OpenMP is easy to use …

```
#include <omp.h>
#include <vector>

int main(){
    std::vector<double> val(1e8,0);
    #pragma omp parallel for
    for (int i = 0; i < val.size(); i++)
        val[i] = COSTLY_OPERATION(i);
    return 0;
}
```

Parallel Region

```
// In Terminal/Command line
// Compile via command line (or makefile)
g++ -fopenmp -O3 main.cpp -o main.exe

// Run
export OMP_NUM_THREADS=2; ./main.exe
```

Parallel computing with **MPI**

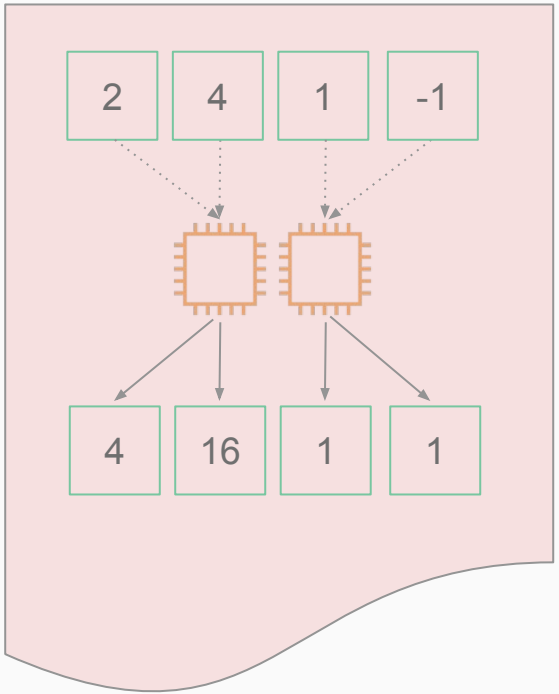
The Message Passing Interface standard was established in mid 90s.
OpenMPI is common implementation of this standard.

Distributed/Shared-Memory Parallelism
Parallelization paradigms

Shared Memory Parallelization

*Computation is distributed along **threads**.*

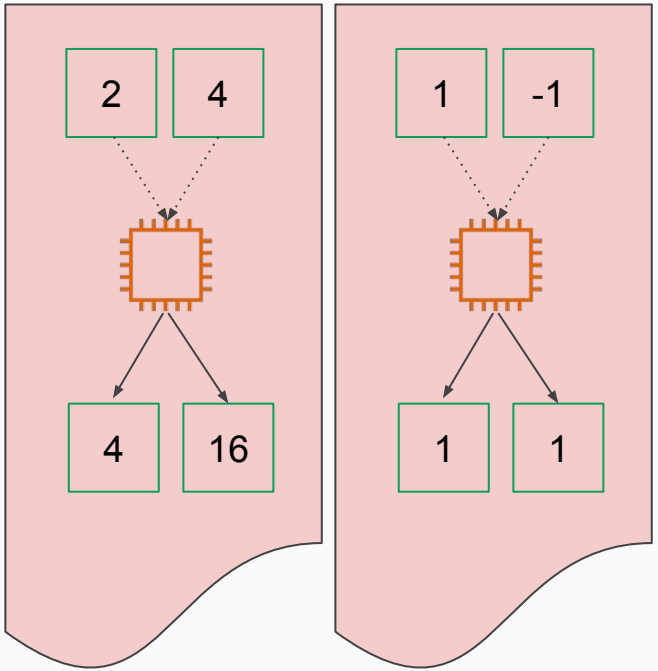
Synchronization between threads.



Distributed Memory Parallelization

*Computation is distributed along **processes**.*

Communication via message passing.



Three Consideration: **code**, **compiler**, and **execution**.

```cpp
#include <mpi.h>
#include <vector>

int main(){
    int size, rank;
    MPI_Init(NULL,NULL); // Needs to be call
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<double> val(size,0);
    val[rank]=some_func(rank);
    MPI_Allreduce( MPI_IN_PLACE,&val[0],size,MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD);
    MPI_Finalize(); // Needs to be called
    return 0;
}
```

Initialize, assign rank and size.
... common start to MPI code.

```
// Compile via command line/makefile
mpic++ -O3 main.cpp -o main.exe

// Run
mpirun -np 2 ./main.exe
```

MPI Programing = **Planning** + **Functions**

Provide control over **program flow** and **communication**.

```
for ( int i = 0; i < n; i++){
 val[i] = some_func(i)
}
```

```
val[rank] = some_func(rank)
MPI_Allreduce( MPI_IN_PLACE,
&val[0],size,MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
```

- **Planning:** How will you split the computation?
- **Communication:** How will you aggregate the results.

MPI requires more planning (*as compared to OpenMP*).

Program Flow

Execute n of the same program

```
#include <mpi.h>
int main(){
   int size, rank;
   MPI_Init(NULL,NULL);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Finalize();
   return 0;
}
```

```
// via terminal/command line
mpirun -np 3 ./main.exe
```

```
#include <mpi.h> //rank=0
#include <mpi.h> //rank=1
#include <mpi.h> //rank=2
int main(){
   int size, rank;
   MPI_Init(NULL,NULL);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Finalize();
   return 0;
}
```

1) **Code:** There is one program/class/cpp file etc.

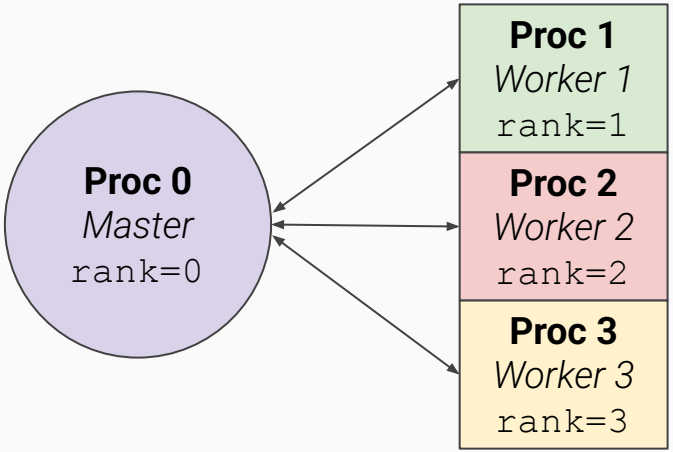
2) **Execution:** `np 3` executions of the program.

3) **Runtime**: each process has a ID, i.e a **`rank`**.

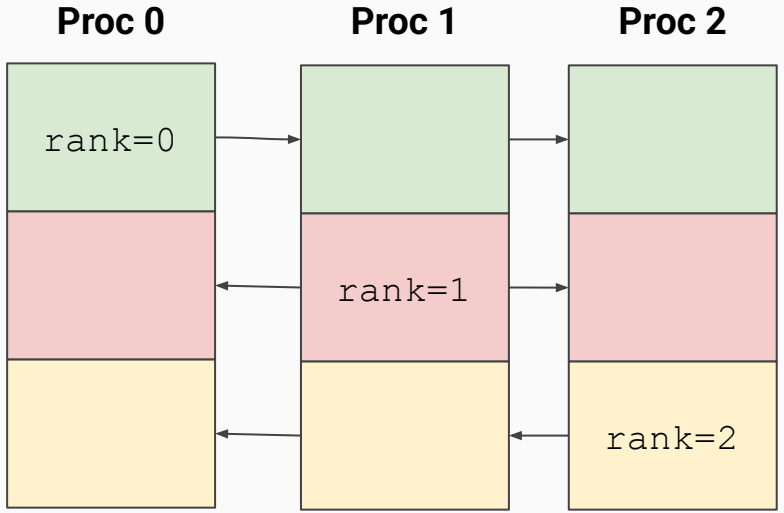
The control of program flow is via **`rank`**.

Program Flow
Some examples (not standards)

Master is the conductor



All processes carry the (full) result



What if the "result" doesn't fit on one process?

Communication
A communication “channel”

`MPI_COMM_WORLD`: The global communication “scope” between processes.

```cpp
#include <mpi.h>
#include <vector>

int main(){
   int size, rank;
   MPI_Init(NULL,NULL); // Needs to be call
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   std::vector<double> val(size,0);
   val[rank]=some_func(rank);
   MPI_Allreduce( MPI_IN_PLACE,&val[0],size,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
   MPI_Finalize(); // Needs to be called
   return 0;
}
```

You can split `MPI_COMM_WORLD` into **groups** (see `MPI_Comm_split`)

groups: Local channels of communication having their own `rank` and `size`.
See here for an excellent tutorial.

All processes within `comm` are required **not** to proceed to the next line of code until **all processes** in `comm` are at this line.

```
int MPI_Barrier( MPI_Comm comm )
```

A parallel operation is, in general, **not synchronized**.

Communication
Point-to-Point

Send **`snd_data`** to rank **`destination`** and
write it to **`rcv_data`** at the rank **`source`**.

```
MPI_Send(void* snd_data,int count,MPI_Datatype datatype,int destination,int tag, MPI_Comm communicator)
MPI_Recv(void* rcv_data,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm communicator,MPI_Status* status)
```

`MPI_Send` and `MPI_Recv` **must be matching**, or else the program hangs!

***Note**: There are asynchronous versions of multiple functions in MPI.*

Communication
Point-to-Point Deadlocks

```
if (rank == 0){
  destination=1;
  source=1;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#1
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#2
}else if(rank == 1){
  destination=0;
  source=0;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#3
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#4
}
```

```
if (rank == 0){
  destination=1;
  source=1;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#1
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#2
}else if(rank == 1){
  destination=0;
  source=0;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#4
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#3
}
```

*Pair **Sends** with **Receives**!*

```
rank 0 → rank 1 #1
rank 1 ← rank 0 #3

rank 0 ← rank 1 #2
rank 1 → rank 0 #4
```

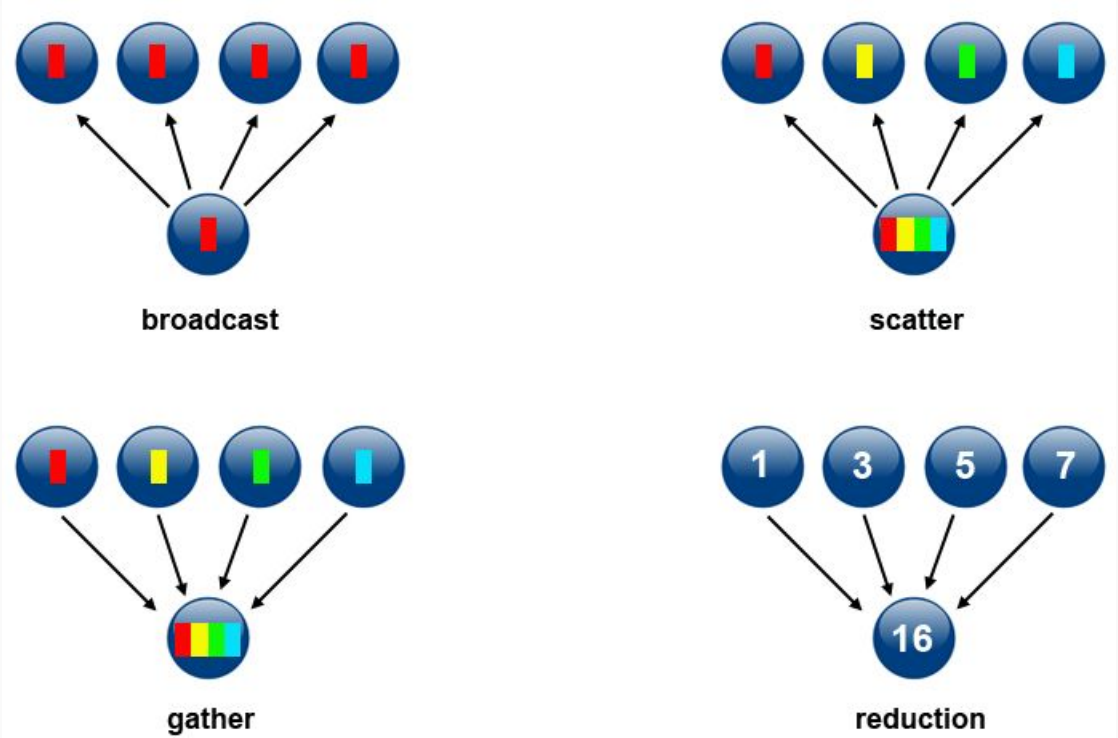
This results in

```
rank 0 → rank 1  #1
rank 1 → rank 0  #4
```

This *could* result in a **deadlock** (not recommended). Both rank 1 and rank 2 send without receiving, and thus wait forever!

Implementation dependent, see https://web.stanford.edu/class/cme194/cgi-bin/lecture5.pdf

Communication
Collective



Source https://hpc-tutorials.llnl.gov/mpi/collective_communication_routines/

For `MPI_Allreduce` and `MPI_Allgather` the results of the base operations are broadcasted i.e., all ranks have the **full** result.

References

Many online reference

See mpitutorial.com for nice examples and references,
and rookiehpc.org for more details on both MPI and openMP.

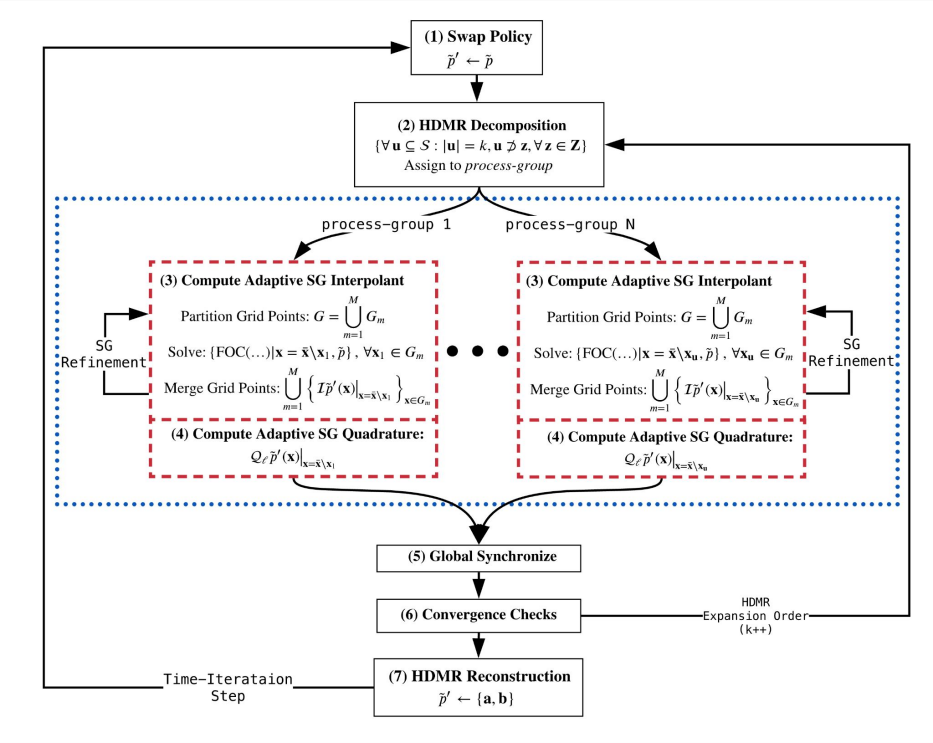
Hybrid MPI and OpenMP Parallel Programming
An example

MPI + OpenMP: Two layers of parallelism

(1) The primary layer uses MPI (dotted blue lines), which ideally splits independent workloads across processes.

(2) Within each workload, the secondary layer using OpenMP (dashed red lines) to perform parallel numerical operations within the workload.

Note the need for “global synchronization” across processes.



E. A., Scheidegger S. Siam SISC, 2022.

Motivation

MPI is designed to scale and

is fundamental for HPC.

Università della Svizzera italiana
Institute of Computing CI

High-Performance Computing, Fall 2024
Lecturer: Dr. A. Eftekhari, Prof. O. Schenk
Assistants: M. Lechekhab, D. Vega,
D. Santarsiero, J. Palumbo, I. Ecevit

Project 4
Parallel Programming using the Message Passing Interface MPI

Due date: 13 November 2024 at 23:59 (See iCorsi for updates)

This assignment will introduce you to parallel programming using the Message Passing Interface (MPI). You will implement a simple MPI message exchange, compute a process topology, parallelize the computation of the Mandelbrot set, and finally implement a parallel matrix-vector multiplication used within the power method. You may do this project in groups of students (max four or five). In fact, we prefer that you do so. Please clearly indicate who you worked with in the "Discussed with" field of your report.

If MPI is new to you, we highly recommend the LLNL tutorial[1]. Likewise, we also highly recommend [3, Chap. 8] and [2, Chap. 1-5][2]. The MPI standard document you find on the MPI forum website[3] is very valuable source of information.

All tests and simulation results must be run on the compute nodes of the Rosa cluster. However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers (the final code must compile and run on Rosa cluster). You will find all the skeleton source codes for the project on the course iCorsi page.

For the whole project, the simulations must be run on the compute nodes of the Rosa cluster, using the GNU Compiler Collection and OpenMPI MPI library:

```
[user@icsnodeXX ~]$ module load gcc openmpi
[user@icsnodeXX ~]$ mpicc -v
Reading specs from ...
COLLECT_GCC= ...
COLLECT_LTO_WRAPPER= ...
Target: x86_64-pc-linux-gnu
Configured with: ...
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 13.2.0 (Spack GCC)
```

However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers, but please make sure to document them in your report if you include results.

While developing/debugging your code, it can be useful to work in an interactive session (you can use the `-reservation=hpc-tuesday` or `-reservation=hpc-wednesday` for better priority) as follows:

```
[user@icslogin01 hello_mpi]$ srun --ntasks=4 --nodes=1 --time=00:05:00
↪  --reservation=hpc-tuesday --pty bash -i
srun: job 12726 queued and waiting for resources
srun: job 12726 has been allocated resources
[user@icsnode33 hello_mpi]$ module load openmpi
[user@icsnode33 hello_mpi]$ make
[user@icsnode33 hello_mpi]$ mpirun hello_mpi # run hello world in skeleton code
Hello world from rank 1 out of 4 on icsnode33
Hello world from rank 3 out of 4 on icsnode33
Hello world from rank 2 out of 4 on icsnode33
Hello world from rank 0 out of 4 on icsnode33
```

[1] https://hpc-tutorials.llnl.gov/mpi/
[2] Both books also feature some advanced topics of interest and you find links to the books on the course iCorsi page.
[3] https://www.mpi-forum.org/

This allocates an interactive session on 1 node of the Rosa cluster with up to 4 tasks (MPI processes). In this allocation, we got all tasks assigned to one single node `icsnode33`. By default, `mpirun` will use all available tasks (here 4). To run with less (e.g., for doing a scaling study), use the `-np N` option, where `N` is the desired number of MPI processes. For example:

```
[user@icsnode33 hello_mpi]$  mpirun -np 2 hello_mpi
Hello world from rank 1 out of 2 on icsnode33
Hello world from rank 0 out of 2 on icsnode33
```

We can also ask Slurm to allocate the job on separate nodes:

```
[user@icslogin01 hello_mpi]$ salloc --ntasks=4 --nodes=4 --time=00:05:00
salloc: Pending job allocation 12776
salloc: job 12776 queued and waiting for resources
salloc: job 12776 has been allocated resources
salloc: Granted job allocation 12776
[user@icslogin01 hello_mpi]$ module load openmpi
[user@icslogin01 hello_mpi]$ srun ./hello_mpi
Hello world from rank 1 out of 4 on icsnode34
Hello world from rank 0 out of 4 on icsnode33
Hello world from rank 2 out of 4 on icsnode35
Hello world from rank 3 out of 4 on icsnode36
```

Please see also the example batch scripts in the `hello_mpi` example of the provided skeleton source codes. Last but not least, the Rosa cluster documentation and the Slurm documentation are essential resources. As usual, you find all the skeleton source codes for the project on the course iCorsi page.

1 Ring sum using MPI [10 Points]

This task familiarizes you with some basic MPI send/receive functionality and identification of the neighbors in a one-dimensional process layout known as a ring topology. The processes are organized in a circular chain along their MPI ranks, where each process has two neighbors and first and last processes are neighbors as well. The ring sum using MPI algorithm proceeds in the following way: every process initially sends its rank number to a neighbor (in increasing rank direction, except for the first and last processes); then every process sends what it receives from that neighbor. This is done n times, where n is the number of processes. As a result, all ranks will accumulate the sum of all ranks. The first two iterations of the algorithm are illustrated in Fig. 1.

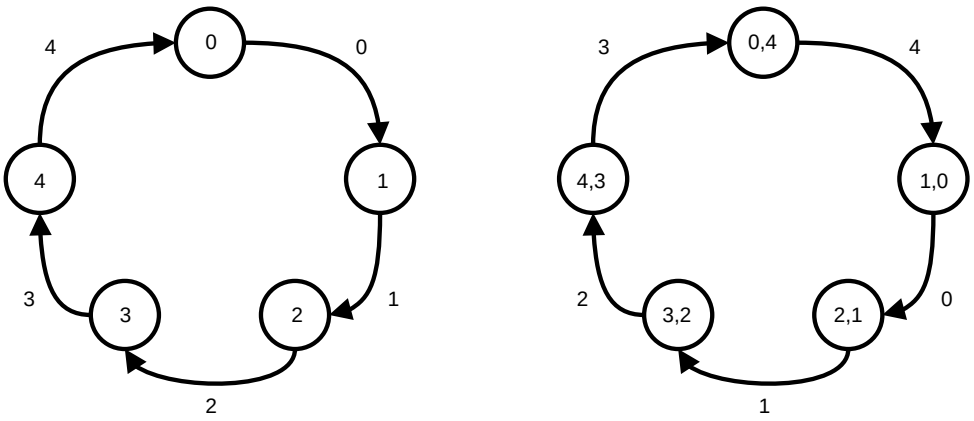


Figure 1: First two iterations of the ring sum algorithm with $n = 5$ MPI processes. The ring sum is equal to 10.

Your task: Implement the ring sum algorithm in the provided skeleton code `ring`. Be careful to avoid any potential deadlock issues in your implementation. In your report, briefly comment on your chosen communication pattern, particularly how you avoid potential deadlock issues.

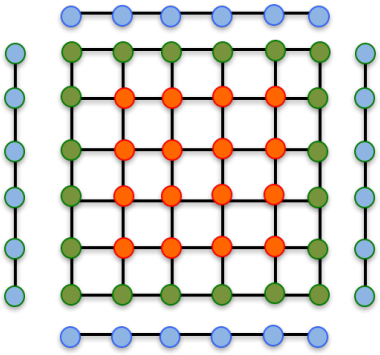
2 Cartesian domain decomposition and ghost cells exchange [15 Points]

The objective of this task is to write an MPI program that partitions a two-dimensional Cartesian domain into a number of sub-domains and to exchange so-called *ghost cells*[4] between neighboring MPI processes. The term ghost cells refers to a copy of remote process' data in the memory space of the current process. This domain decomposition and ghost cells exchange is extensively used in stencil-based kernels such as in the mini-app solving Fisher's equation in Project 3 need data from grid cells residing on another process.

In this task, we consider a two-dimensional and periodic toy domain discretized by 24×24 grid points. This domain is distributed over a 4×4 "grid of processes". Each process holds a local domain discretized by 6×6 grid points, which is extended by one row/column on each side in order to accommodate the copy of its neighbors' borders, i.e., the ghost cells. For simplicity, we ignore the corner cells. The grid of processes and a local domain is illustrated in Fig. 2. The processes are arranged in a so-called Cartesian topology with periodic boundaries, which means that, for example, process with rank 0 is also a neighbor of processes with rank 3 and 12. Therefore, each process has four neighbors and these are often referred to the north, south, east and west neighbor. The exchange of ghost cells between processes of rank 5 and 9 is illustrated in Fig. 3.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Domain distributed over 4×4 "grid of processes".



(b) Local domain discretized by 6×6 grid points extended by one row/column of ghost cells on each side.

Figure 2: Cartesian domain decomposition.

We provide a skeleton code in the `ghost` directory and its compilation and execution on the Rosa cluster are shown below:

```
[user@icslogin01 ghost]$ srun --ntasks=16 --pty bash -i
[user@icsnodeXX ghost]$ module load openmpi
[user@icsnodeXX ghost]$ make
[user@icsnodeXX ghost]$ mpirun -np 16 ./ghost
```

Note: If you are developing on your own machine (or a machine with less than 16 processors), you will need to *oversubscribe* your processors. With OpenMPI, this is achieved with flag `--oversubscribe` (see here for details).
The result of the boundary exchange on rank 9 should be

```
9.0  5.0  5.0  5.0  5.0  5.0  5.0  9.0
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
```

[4] The terms *guard* or *halo cells* is also used.

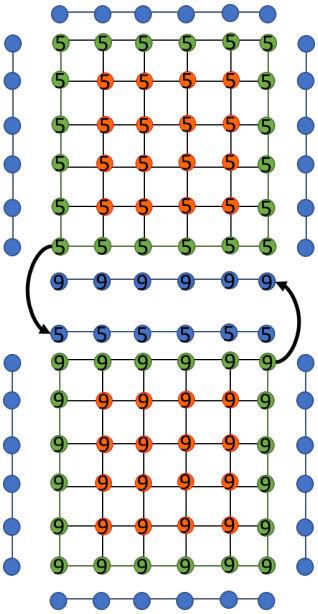


Figure 3: Exchange of ghost cells between processes of rank 5 and 9.

```
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
9.0 13.0 13.0 13.0 13.0 13.0 13.0  9.0
```

Starting from the provided skeleton code, complete the following tasks:

1. Create a Cartesian two-dimensional MPI communicator (4×4) with periodic boundaries and use it to find your neighboring ranks in all dimensions in a cyclic manner.
   Hint: Use `MPI_Cart_create` and related functions.

2. Create a derived data type for sending a column boundary (east and west neighbors).
   Hint: Use `MPI_Type_vector` and related functions.

3. Exchange ghost cells with the neighboring cells in all directions and verify that correct values are in the ghost cells after the communication phase. Be careful to avoid any potential deadlock issues in your implementation.

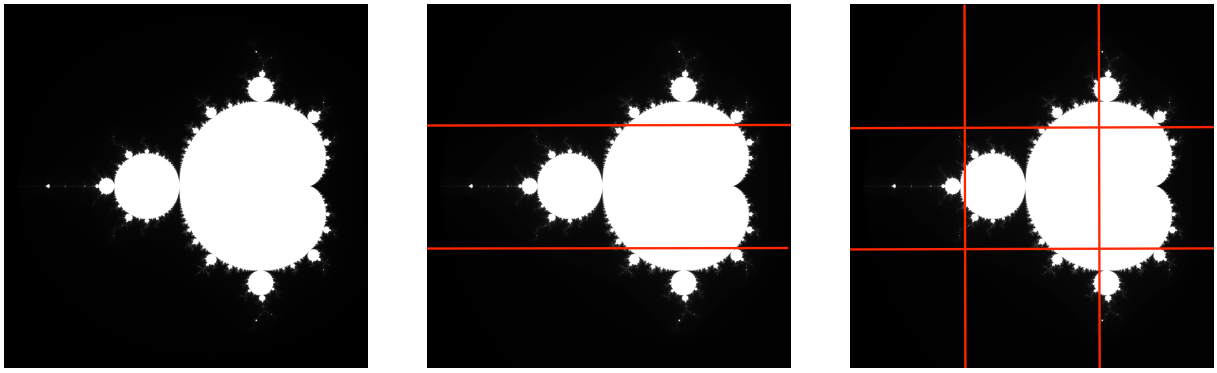
4. **Bonus [10 Points]**: Also exchange ghost values with the neighbors in ordinal directions (northeast, southeast, southwest and northwest).

3 Parallelizing the Mandelbrot set using MPI [20 Points]

The goal of this task is to parallelize the Mandelbrot set computation from Project 2 using MPI. The computation of the Mandelbrot set will be partitioned into a set of parallel MPI processes, where each process will compute only its local portion of the Mandelbrot set. Examples of a possible partitioning are illustrated in Fig. 4. After each process completes its own computation, the local domain is sent to the master process that will handle the I/O and create the output image containing the whole Mandelbrot set.

We introduce two structures, that represent the information about the partitioning. These structures are defined in `consts.h`. The structure `Partition` represents the layout of the grid of processes and contains information such as the number of processes in x and y directions and the coordinates of the current MPI process:

```c
typedef struct {
    int x;          // x-coordinate of current MPI process in the process grid
    int y;          // y-coordinate of current MPI process in the process grid
    int nx;         // Number of processes in x-direction
```



(a) One partition. (b) Three partitions. (c) Nine partitions.

Figure 4: Possible partitioning of the Mandelbrot set.

```
    int ny;         // Number of processes in y-direction
    MPI_Comm comm; // (Cartesian) MPI communicator
} Partition;
```

The second structure `Domain` represents the information about the local domain of the current MPI process. It holds information such as the size of the local domain (number of pixels in each dimension) and its global indices (index of the first and the last pixel in the full image of the Mandelbrot set that will be computed by the current process):

```
typedef struct {
    long nx;     // Local  domain size in x-direction
    long ny;     // Local  domain size in y-direction
    long startx; // Global domain start index of local domain in x-direction
    long starty; // Global domain start index of local domain in y-direction
    long endx;   // Global domain end   index of local domain in x-direction
    long endy;   // Global domain end   index of local domain in y-direction
} Domain;
```

The skeleton code you find on iCorsi is initialized in a way that each process computes the whole Mandelbrot set. Your task is to partition the domain (so that each process only computes an appropriate part), compute the local part of the image, and send the local data to the master process that will create the final complete image. The compilation and execution on the Rosa cluster are shown below:

```
[user@icslogin01 mandel]$ srun --ntasks=16 --pty bash -i
[user@icsnodeXX mandel]$ module load openmpi
[user@icsnodeXX mandel]$ make
[user@icsnodeXX mandel]$ mpirun -np 16 ./mandel_mpi
```

Running the provided benchmark script and creating the performance plot is achieved by submitting the job to the Slurm queuing system. The runtime configuration controling the number of nodes and MPI ranks is specified in the `run_perf.sh` script:

```
[user@icslogin01 mandel]$ module load openmpi
[user@icslogin01 mandel]$ make
[user@icslogin01 mandel]$ sbatch run_perf.sh
```

This creates a graph `perf.pdf` showing the wall-clock time of each process for multiple runs with varying number of processes ($N_{\text{CPU}} = 1, 2, 4, 8, 16$).

Solve the following tasks:

1. Create the partitioning of the image by implementing the functions `createPartition` and `updatePartition`. You can find a dummy implementation of these functions in `consts.h`.

2. Determine the dimensions and the start/end of the local domain based on the computed partitioning by implementing the function `createDomain`. The function is defined in the `consts.h`.

3. Send the local domain to the master process if `mpi_rank > 0` and receive it at the master process where `mpi_rank == 0`. Compare the output of the parallelized program to that of the sequential program in a graphic and verify that it is correct.

4. Comment the performance observed in the graph `perf.pdf` in your report. Give a suggestion to improve the performance[5].

4 Parallel matrix-vector multiplication and the power method [40 Points]

This task is about writing a parallel program to multiply a matrix A by a vector x, and to use this routine in an implementation of the power method[6] to find the largest absolute eigenvalue of a given matrix. A serial `Python` implementation of the power method is provided in the skeleton codes `powermethod.py`. A less experienced parallel programmer has already started the implementation in the skeleton `powermethod_rows.c` and your task is to complete the parallelization. The skeleton code provides four test cases and the exact result is known for the first three cases. Use them to verify your code.

The next task is to study the parallel scalability of your implementation. To this end, use test case three `test_case = 3`, set the matrix size to `n = 10'000` and fix the maximum number of iterations to `niter = 300`. Fix the tolerance to a negative value to make sure that earlier convergence does not interfere with the timing measurements (e.g., `tol = -1e-6`). Do the following parallel scaling studies:

1. **Strong scaling**: Run you code for $p = 1, 2, 4, 8, 16, 20$ MPI processes. Plot the runtime and the parallel efficiency as a function of the number of MPI processes.

2. **Weak scaling**: Run you code for $p = 1, 2, 4, 8, 16, 20$ MPI processes and make the matrix size n grow (nearly) proportional to $\sqrt{p}$. Since both total memory and total work scale as n^2, this implies that the memory required per processor and the work done per process will remain constant as you increase p. Plot the runtime and the parallel efficiency as a function of the number of MPI processes/problem size.

Perform the strong and weak scaling study for two MPI process distributions: (i) all MPI processes on one node, and (ii) all MPI processes on different nodes. In your report, provide a short description and interpretation of your parallel scaling studies (and don't forget to include and reference the figures).

5 Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - Follow the provided guidelines for the report.
- Submit your .tgz through iCorsi.

[5] You don't have to implement your suggestion.
[6] See, e.g., [1, Algorithm 4.1] or https://en.wikipedia.org/wiki/Power_iteration for more information.

Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

References

[1] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. Society for Industrial and Applied Mathematics, January 2000. doi:10.1137/1.9780898719581.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. MIT Press Ltd, November 2014. URL: https://ieeexplore.ieee.org/servlet/opac?bknumber=6981847.

[3] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: http://dx.doi.org/10.1201/EBK1439811924, doi:10.1201/ebk1439811924.

Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 4

1. Task: Ring maximum using MPI [10 Points]

There are multiple possible solutions to solve this problem without having a deadlock. I chose to separate my processes into two disjointed groups based on their rank id. More precisely into even and odd ranks, where they alternate between sending and receiving and therefore preventing deadlocks. In this implementation even rank ids send in a first step and then receive in a second step, the odd rank ids do the opposite. For an even number of total processes this works because the highest rank id is odd and therefore for a given process all its neighbors are sending while the particular process is receiving and vice versa. When having an odd number of total processes, this results in the highest rank id being even and wanting to send their content to rank 0. Even though both processes are sending at the same time, based on the given topology we know that process 0 will first send it contents to an odd rank process and is then able to receive the contents from the highest rank process.

```
int sum = rank;
int rec_rank = rank;
int send_rank = rank;
for (int i = 0; i < size - 1; i++) {
  if (rank % 2 == 0) {
    MPI_Send(&send_rank, 1, MPI_INT, (rank + 1) % size, tag, MPI_COMM_WORLD);
    MPI_Recv(&rec_rank, 1, MPI_INT, (rank - 1 + size) % size, tag,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(&rec_rank, 1, MPI_INT, (rank - 1 + size) % size, tag,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&send_rank, 1, MPI_INT, (rank + 1) % size, tag, MPI_COMM_WORLD);
  }
  sum += rec_rank;
  send_rank = rec_rank;
}
```

Listing 1: Ring topology using alternating send and receive

2. Task: Ghost cells exchange between neighboring processes [15 Points]

For this exercise we create a Cartesian communicator with periodic boundaries, in order to find the neighboring ranks using the `MPI_Cart_Shift` function. After finding all four neighbors we do four

`MPI_Sendrecv` executions, where each processes, for example receives the values from the southern neighbor and sends its values to the northern neighbor, which is then repeated for each direction. Note that this is possible because we have periodic boundaries which means every processes has a neighbor in every direction.

```
// Create cartesian topology
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_cart);

// Finding neighbors
MPI_Cart_shift(comm_cart, 0, 1, &rank_top, &rank_bottom);
MPI_Cart_shift(comm_cart, 1, 1, &rank_left, &rank_right);

// Send values to northern neighbor and receive from southern neighbor
MPI_Sendrecv(&data[1 * DOMAINSIZE + 1], 1, T_row, rank_top, 0,
       &data[(SUBDOMAIN + 1) * DOMAINSIZE + 1], 1, T_row, rank_bottom,
       0, comm_cart, MPI_STATUS_IGNORE);

/* REPEATED FOR EACH DIRECTION */
```

Listing 2: Example

In order to simply the communications, we introduce two new MPI data types. For the row we define a `MPI_Type_contiguous`, which is simply for values that are stored in a contiguous location. For the columns we define `MPI_Type_vector` data types, which allows us to define a stride and therefore columns in a contiguous stored matrix.

```
MPI_Datatype T_row, T_col;
MPI_Type_contiguous(SUBDOMAIN, MPI_DOUBLE, &T_row);
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &T_col);
MPI_Type_commit(&T_col);
MPI_Type_commit(&T_row);
```

Listing 3: Example

For the exchange of ghost cells of the neighbors in ordinal direction, we use the `MPI_Cart_coords` function, which returns the coordinates of a process in a communicator of a cartesian topology. By adding $+1$ or -1 to its own x and y coordinates, a process can figure out its ordinal neighbors as seen in Listing 4. Finally we do a send and receive similarly as before, just in ordinal direction. For example sending the value to the northeast and receiving the values from southwest.

```
int top_right_rank, top_left_rank, bottom_right_rank, bottom_left_rank;
int top_right_coords[2] = {coords[0] - 1, coords[1] + 1};
MPI_Cart_rank(comm_cart, top_right_coords, &top_right_rank);
int top_left_coords[2] = {coords[0] - 1, coords[1] - 1};
MPI_Cart_rank(comm_cart, top_left_coords, &top_left_rank);
int bottom_right_coords[2] = {coords[0] + 1, coords[1] + 1};
MPI_Cart_rank(comm_cart, bottom_right_coords, &bottom_right_rank);
int bottom_left_coords[2] = {coords[0] + 1, coords[1] - 1};
MPI_Cart_rank(comm_cart, bottom_left_coords, &bottom_left_rank);

// Send top right and receive from bottom left
MPI_Sendrecv(&data[1 * DOMAINSIZE + (DOMAINSIZE - 2)], 1, MPI_DOUBLE,
               top_right_rank, 0, &data[(DOMAINSIZE - 1) * DOMAINSIZE + 0], 1,
               MPI_DOUBLE, bottom_left_rank, 0, comm_cart, MPI_STATUS_IGNORE);

/* REPEATED FOR EVERY ORDINAL DIRECTION */
```

Listing 4: Example

```
data of rank 9 after communication
 4.0  5.0  5.0  5.0  5.0  5.0  5.0  6.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
12.0 13.0 13.0 13.0 13.0 13.0 13.0 14.0
[huberde@icsnode30 ghost]$ _
```

Figure 1: Result for process with Rank 9

3. Task: Parallelizing the Mandelbrot set using MPI [20 Points]

In 2 you can see the performance of the Mandelbrot set spread out over different number of processes. This shows how much work each thread has to do depending on the number of processes. If we have more than two processes we can clearly see that the workload is unevenly distributed across the processes. This imbalance is caused by the fact that different regions of the complex plane need varying numbers of iterations to determine if a point is part of the mandelbrot set. For higher number of processes we can see that workload is distributed more evenly, yet there are still processes that do very little compared to others.

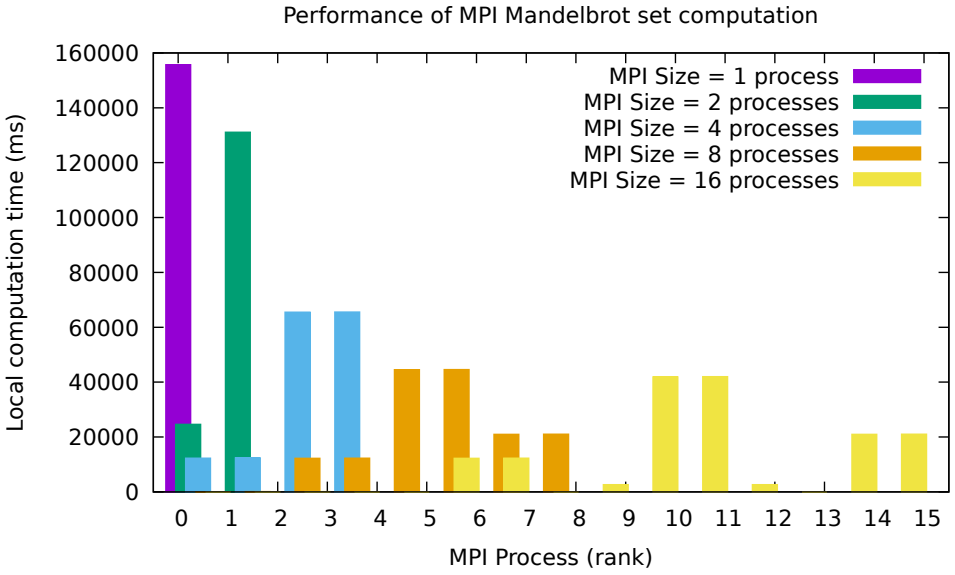


Figure 2: MPI Performance of the Mandelbrot on the Rosa Cluster for 1 to 16 Processes

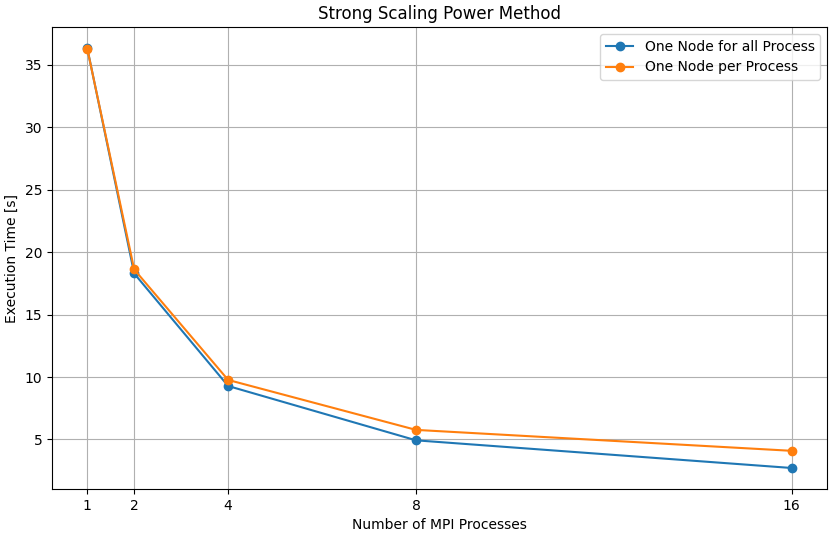
In order to improve the performance a different partitioning scheme with different sized grids could be used. I would make the number of grids higher in computational demanding areas and as a result decrease the size of this grids in said area. Furthermore the grids would be bigger for less computational demanding areas. This would result in an improved workload balanced across all the processes.

4. Task: Parallel matrix-vector multiplication and the power method [40 Points]

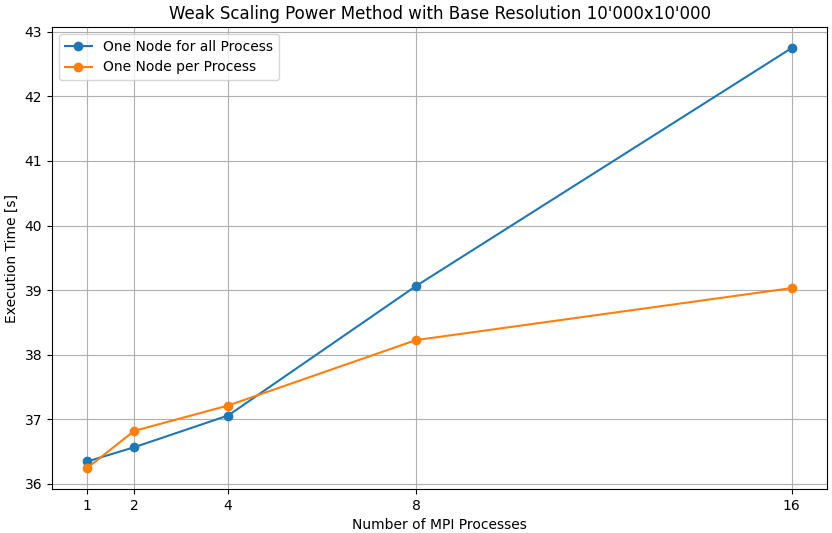
For both, strong and weak scaling Test case 3 was used where $A(i,j) = i$ if $i == j$ for $1 <= i,\ j <= n$ and the eigenvalue should converge to $\theta = n$, the dimension of the quadratic matrix.

Furthermore the iteration were capped to 300 iterations and no tolerance limit was set, therefore in the case of early converges, the calculation would continue. For the strong scaling we used the fixed matrix size of $n = 10000$. This size was also used as the base matrix size for the weak scaling version, which was then multiplied by $\sqrt{p}$, where p is the number of MPI processes.

When comparing the execution time between running the MPI processes on a single node or distributed across multiple nodes, in the strong scaling case, seen in Subfigure 3a, we can see that running the power method on a single or multiple are basically identical. There's a strong increase in performance for a low number of processes, which then later becomes less when the overhead increases. It's slightly diverging for higher number of processes we can be caused by other users running their program on the server or the communication overhead that needs to travel a larger distance.



(a) Strong Scaling



(b) Weak Scaling

Figure 3: Scaling Plots of the Power Methods

In the weak scaling case in Subfigure 3b where we increase the matrix size proportionally with the number of processes. Here the optimal case would be that the execution time stays constant, but we can clearly see that single node version, scales significantly worse, compared to running the

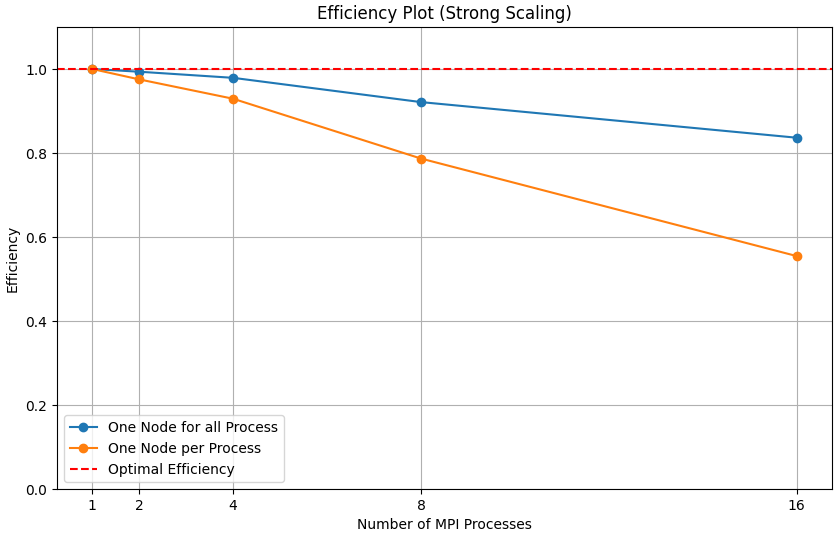
power method on multiple nodes. This caused that more and more resources are contested on a single node, compared to running the program on multiple nodes. This impact can be especially felt with a higher number of processes, where by definition the matrix size is significantly bigger. Hence the execution time on a single node increase a lot stronger on a single node.

Looking at the efficiency plots (4) they paint a similar picture than the previous plots. For the strong scaling case, the efficiency is given by

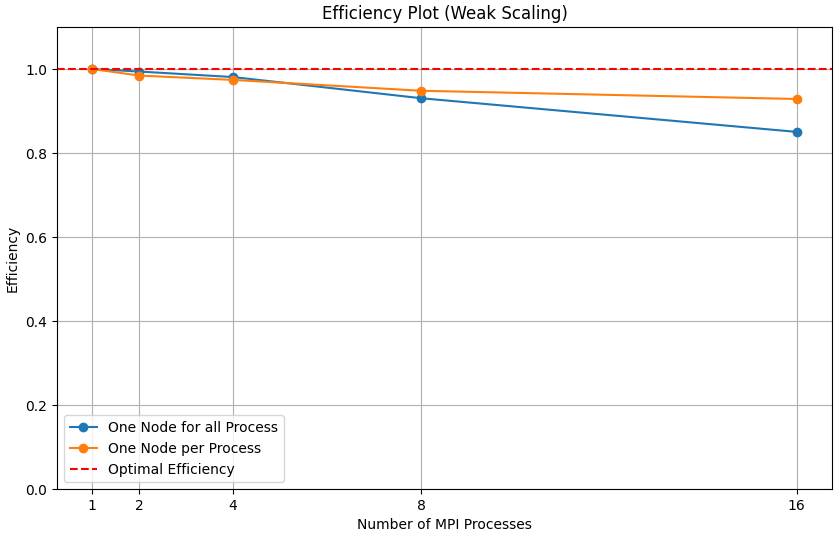
$$E_S = \frac{T_1}{T_p \cdot p} \tag{1}$$

For the weak scaling the efficiency formula slight differs and is given as follows

$$E_W = \frac{T_1}{T_p} \tag{2}$$



(a) Strong scaling



(b) Weak Scaling

Figure 4: Efficiency Plots for the Weak and Strong Scaling case

In the strong scaling case in Subfigure 4a, for smaller number of processes the efficiency is relatively high and then decreases for higher number of processes due to the overhead by using more processes. The efficiency drops strong for the power method run on multiple nodes compared to the one run on a single node, this again highlights the increased overhead created when communication between multiple nodes.

For the weak scaling case in Subfigure 4b, due to the proportionally increased workload per number of processes the efficiency is clearly better than in Strong scaling case. Due to increasing size of the matrix the overhead to workload ratio is lower, resulting in higher efficiency. Comparing the single to the multiple node, we can see that the power method run on multiple nodes is slightly more efficient, especially for a higher number of processes.