Exercises for High Performance Computing
(MA-INF 1106)
WS 2023/2024

E. Suarez, M. Wolter and B. Kostrzewa
Tutors: O. Vrapcani and N. Pillath

---

10 Distributed-memory parallelism using MPI

ATTENTION!: eCampus can become unavailable without previous announcement due to an urgent maintenance. Students are responsible for submitting their checklists enough ahead of time of the deadline. Submissions via Email will not be accepted unless tutors explicitly authorized it beforehand.

This exercise can be performed on your own machine as long as you are able to install some kind of MPI implementation. Failing that you can rely on the **JURECA-DC system**, but be very mindful of computing time.

WARNING: Be very conscious about compute time. **Run your jobs only in batch mode**, unless the exercise explicitly asks to run an interactive session. When you run in interactive mode, select your time windows only as long as required to run the job and close the interactive session immediately after finishing. If you use Jupyter, run your Jupyter notebook only on the login node. **Do NOT start Jupyter on the compute node** because these long sessions consume too much compute time. Instead, to start an interactive session from Jupyter running on the login node, just open a terminal from the Jupyter launcher.

Important recommendations for benchmarking runs:

- Try first with just the first 4-5 smallest vector lengths until you are sure all your scripts are doing what they should.
- Include in the beginning of your output files a header containing in a few lines all important information of the run conditions, e.g.: name and version of source code, compiler version and flags used, node number on which it run, number of threads and/or processes, etc.

Unless you choose to submit manually, please use `https://classroom.github.com/a/Ib03X1s9` to submit your work.

Introductory remarks

In this exercise sheet we will explore distributed-memory parallelism using a number of simple MPI constructs to parallelise dense matrix-vector multiplication in different ways. MPI is a rather complex topic and we will only be able to cover a very small fraction of the standard in this problem sheet. With a few important exceptions we will aim to give you (almost) all of the tools that you will need for implementing simple parallel algorithms using message passing.
The manual pages of the MPI implementation *MPICH*, available at `https://www.mpich.org/static/docs/latest/` will be an important (if partly incomplete) resource. Another important resource is the RookieHPC website at `https://rookiehpc.org/`. In this problem sheet we will encounter the following functions and data structures:

- MPI_Init
- MPI_Initialized
- MPI_Finalize
- MPI_Send
- MPI_Recv
- MPI_Sendrecv
- MPI_Isend
- MPI_Irecv
- MPI_Waitall
- MPI_Gather
- MPI_Allgather
- MPI_Reduce
- MPI_Bcast
- MPI_Allreduce
- MPI_Wtime
- MPI_Comm_split
- MPI_Status
- MPI_Request

Important omissions include:

- MPI_Cart_create
- MPI_Graph_create

- MPI_Dist_graph_create
- MPI_Alltoall
- MPI_Scatter
- MPI_Put
- MPI_Get
- MPI_Test

We will also not be able to explore the intricacies of MPI process distribution through `sbatch` / `srun` or `mpirun`, the complexity induced by the combination of using MPI in a multithreaded program and the important topic of parallel data handling via MPI-I/O.

Compiling MPI applications

MPI implementations use a compiler wrapper to wrap the underlying compiler and to hide the complexity (if any) of providing the correct include paths or linking information for the MPI headers or libraries (if any). Different MPI implementations may use different names for their compiler wrappers and a single MPI implementation may support multiple underlying compilers which may have to be called using a differently named compiler wrapper.
Using these compiler wrappers one can compile MPI applications mostly as usual:

```
$ mpicc source.c -o executable
```

Common examples of compiler wrappers:

- OpenMPI with GCC (`gcc`, `g++`): `mpicc` (C) and `mpicxx` (C++)
- IntelMPI with the Intel compiler (classic, `icc`, `icpc`): `mpiicc` and `mpiicpc`
- IntelMPI with the OneAPI Intel compiler (OneAPI, `icx`, `icpx`): `mpiicc` and `mpiicpc`
- CrayMPICH with GCC (`gcc`, `g++`): `cc` (C) and `CC` (C++)

The software stacks offered by different computing centers provide modules which load these compiler wrappers and set necessary compiler variables. For example, in `Stages/2024` on JURECA-DC, we have `OpenMPI/4.1.5` on top of `GCC/12.3.0`. To load this combination, we have to do:

```
$ module load Stages/2024 GCC/12.3.0 OpenMPI/4.1.5
```

and it may be that the MPI module can be used with multiple underlying compilers.

Launching MPI applications

On your own machine

Almost all of the exercises in this problem sheet can be done on your own machine.

- Install an OpenMPI package provided by your Linux distributon.
- Start the program using `mpirun -np N --oversubscribe ${exe}`, where `N` is the number of MPI ranks to use and `--oversubscribe` allows MPI to launch more processes than the number of hardware threads available on your machine.

On (most) HPC systems running Slurm

For example, on JURECA-DC (2x64 cores per node), to launch a pure MPI program with 256 MPI tasks on two nodes with 128 tasks per node and to place just a single MPI task on each physical core, we would do something like:

```
#!/bin/bash
#SBATCH --time=00:10:00
#SBATCH --job-name=name_of_the_job
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --ntasks-per-node=128
#SBATCH --cpus-per-task=1
#SBATCH --hint=nomultithread
#SBATCH --threads-per-core=1
#SBATCH --error=log.%x_%j.err
#SBATCH --output=log.%x_%j.out

srun --cpus-per-task=1 --hint=nomultithread --threads-per-core=1 ${exe}
```

For a hybird application (MPI + OpenMP) with 8 MPI tasks per node (corresponding to the 8 NUMA regions on the node) and 16 OpenMP threads per MPI task to use all available cores, we would do something like:

```
#!/bin/bash
#SBATCH --time=00:10:00
#SBATCH --job-name=name_of_the_job
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=16
#SBATCH --hint=nomultithread
#SBATCH --threads-per-core=1
#SBATCH --error=log.%x_%j.err
#SBATCH --output=log.%x_%j.out

OMP_NUM_THREADS=16
```

```
OMP_PROC_BIND=close
OMP_PLACES=cores

srun --cpus-per-task=16 --hint=nomultithread --threads-per-core=1 ${exe}
```

And if we wanted to additionally use all hyperthreads:

```
#!/bin/bash
#SBATCH --time=00:10:00
#SBATCH --job-name=name_of_the_job
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=32
#SBATCH --hint=multithread
#SBATCH --threads-per-core=2
#SBATCH --error=log.%x_%j.err
#SBATCH --output=log.%x_%j.out

OMP_NUM_THREADS=32
OMP_PROC_BIND=close
OMP_PLACES=threads

srun --cpus-per-task=32 --hint=multithread --threads-per-core=2 ${exe}
```

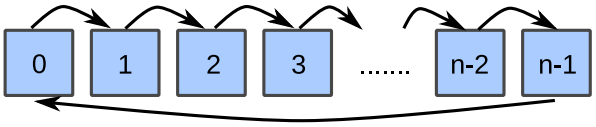
Futher parameters exist to control how exactly the MPI tasks are distributed (mapped) to the machine. Depending on the network topology, this may have a large effect on performance.

1: Basic concepts (7 points)

Tasks:

a) **[1 pt]** Initialize MPI and check that the initialisation was successful. Output the ID of the MPI rank and the total number of ranks.

b) **[1 pt]** Send an integer to `rank+1` and receive an integer from `rank-1` with periodic boundary conditions using a pair of `MPI_Send` and `MPI_Recv`.



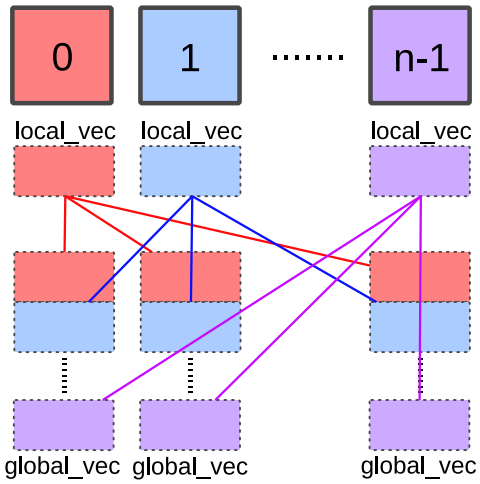
Note that the `tag` argument is arbitrary and can be used to match specific `Send` and `Recv` operations.

c) **[1 pt]** Because this kind of paired sending and reciving is quite typical, MPI offers the compact `MPI_Sendrecv`. Use it to repeat the previous exercise.

d) **[1 pt]** In some situations pairing `Send` and `Recv` operations may lead to deadlocks. In addition, for maximum performance you want to *overlap* communication and computation whenever possible. One way would be to issue `Recv` and `Send` calls from one or multiple threads while other threads perform calculations on a subset of the computational volume which does not require communication. The other way is to make use of the non-blocking (or *immediate*) `MPI_Isend` and `MPI_Irecv`, which return immediately after scheduling an `MPI_Request`. These requests then need to be waited on, which we will do using `MPI_Waitall`. In addition to the array of requests, this function requires an array of `MPI_Status` to be passed into which it writes status information for each request.[1] We are going to make use of a loop over rank IDs and `MPI_Barrier` to order to output.

e) **[1 pt]** On each rank, we will now initialize a vector `local_vec` of size `local_vec_size` elements using random numbers with a rank-dependent seed. We then want to make this data available on all ranks using `MPI_Allgather` in an array called `global_vec`. In order to check if the `Allgather` was sucessful and to understand how the data has been ordered, we use explicit `MPI_Send` and `MPI_Recv` pairs: rank 0 will issue a `Recv` while all other ranks will issue `Send` (in a loop over all ranks > 1). On rank 0 we will then make use of the function `compare_gather_remote` if the data received in the two ways is identical.

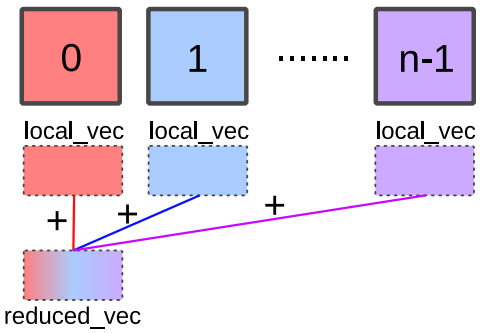
[1] Note: even though `MPI_Isend` and `MPI_Irecv` are often claimed to enable "communication in the background", this is not the case in practice with most MPI implementations. Instead, explicit waiting from one or multiple threads or calls to `MPI_Test` may be necessary to actually progress the communication operations.



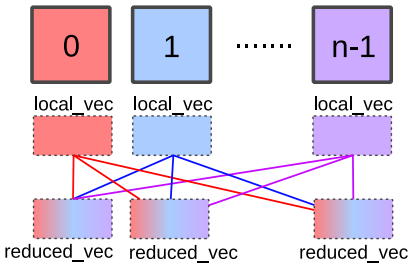
f) **[2 pt]** We will repeat the random initialisation of the local vectors and instead of distributing them using `MPI_Allgather`, we will issue a series of `MPI_Isend` and `MPI_Irecv` in a loop over all ranks not including the rank of the current process. We will again verify that the data exchanged is consistent. Instead of doing this just on rank 0, we will check on all ranks.

2: Reductions (3 points)
Tasks:

a) **[2 pt]** Reductions (sums, products, minima, maxima, combinatorial operations etc.) are an important class of collective communications and often a key bottleneck. We again initialise a random vector `local_vec` using a rank-dependent seed and perform an elementwise sum of the vectors of all ranks on rank 0 using `MPI_Reduce` and the operation `MPI_SUM`. We verify using explicit `Send` and `Recv` and a manual accumulation that the reduction was performed correctly. We broadcast the result of this check from rank 0 to all other ranks using `MPI_Bcast`.

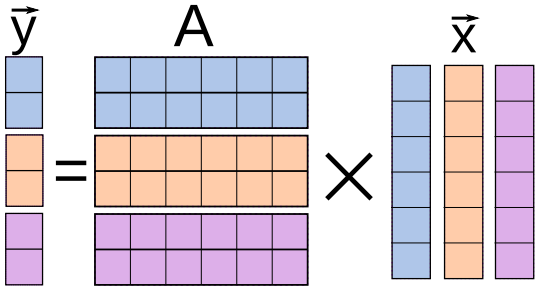


b) **[1 pt]** Instead of performing a reduction just on rank 0 we do so on all ranks using `MPI_Allreduce`.



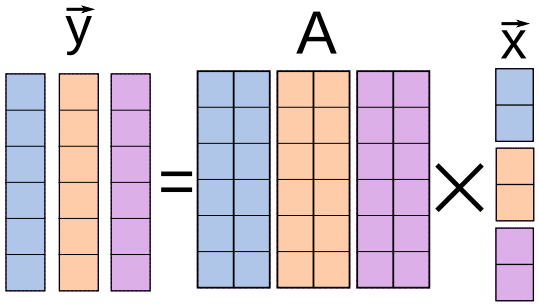
3: Distributed-memory dense matrix-vector multiplication (11 points)
Tasks:

a) **[2 pt]** We used the matrix vector multiplication (MVM), $\vec{y} = A\vec{x}$, as an example in a large number of previous exercises and we are going to take it up here again. Imagine that you're in a situation in which you have to multiply a very large square matrix with a vector, so large in fact, that it does not fit into memory on a single node. One approach would be to slice the matrix rowwise and perform the multiplications and summations for a number of rows per MPI rank on different compute nodes. To do this, the full $\vec{x}$ would of course need to be available on all MPI ranks, correctly initialised. In fact, instead of simply performing $\vec{y} = A\vec{x}$, we want to compute $\vec{y}_n = A^n\vec{x}$, such that the output of the $(n-1)^{\text{th}}$ multiplication is the input of the n^{th} multiplication. In order to achieve this, we will have to gather $\vec{y}_{n-1}$ from all MPI ranks for each iteration n using `MPI_Allgather`.

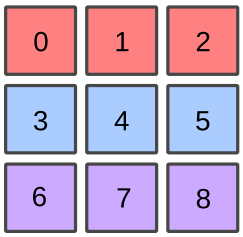


In the exercise we are of course only simulating this: we generate the full matrix A on all ranks an assign only the relevant subset to `A_local`. For testing purposes we also perform the calculation serially and then compare with our parallel result. We measure the time per iteration in multiple attempts and take the best time as our result. In an MPI-parallel application, the different MPI ranks might need slightly differing amounts of time. Use `MPI_Allreduce` to take the minumum value over all MPI ranks.

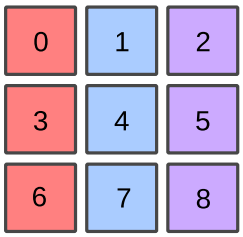
b) **[3 pt]** An alternative might be to instead slice the matrix columnwise and perform submultiplications on each MPI rank. To get the final result we need to sum up the individual $\vec{y}$ from all MPI ranks. Since we want to compute $\vec{y}_n = A^n\vec{x}$, we will need to select the right slice of $\vec{y}_{n-1}$ at each iteration as input for the next.



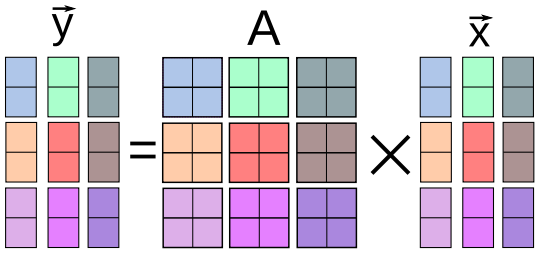
c) **[1 pt]** We have so far exclusively relied on the `MPI_COMM_WORLD` global communicator which is used to launch communications within the set of all N MPI ranks. If we want to perform communication within subsets of these, we can use `MPI_Comm_split` to split an existing communicator into G groups (also referred to as colors), creating N/G new communicators. For each rank n in the old communicator we need to specify which color this rank belongs to as well as a rank ID within the new communicator to be created. Imagine the ranks in a 2D grid and split them rowwise, assigning `ranks_per_color` ranks to the same color. Use `MPI_Comm_rank` and `MPI_Comm_size` to confirm that your new communicators have the right size and that the rank assignments within them are what you expect.



d) **[1 pt]** We can split a communicator into as many different sub-communicators as we like. Split `MPI_COMM_WORLD` rowwise and columnwise, taking `n_ranks_per_row` as a basis for the rowwise coloring and rank assignments. The columnwise assignment follows in an obvious manner.



e) **[4 pt]** We will now combine rowwise and columnwise distribution of the matrix and slice it in both dimensions. To do so we will have to duplicate the right slices of $\vec{x}$ on subsets of MPI ranks and perform gather and reduction operations in the two dimensions on different subsets of MPI ranks, which we will construct as in the previous two tasks by splitting `MPI_COMM_WORLD` rowwise and columnwise.



Commit your solutions to the GitHub Classroom.
If you have used Jupyter, close your Jupyter session and stop JupyterLabs.