Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Benedict Armstrong Discussed with: Tristan Gabl

Solution for Project 5

Due date: Monday 13 May 2024, 23:59 (midnight).

1. Introduction

All benchmarks and programs were run on the **Euler VII — phase 2 cluster** with **AMD EPYC 7763** cpus.

2. Parallel Space Solution of a nonlinear PDE using MPI [in total 60 points]

2.1. Initialize/finalize MPI and welcome message [5 Points]

Changed the welcome message to include the number of processes. The message now reads:

```
================================================================================
                      Welcome to mini-stencil!
version   :: C++ MPI
processes :: 16
mesh      :: 100 * 100 dx = 0.010101
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
```

2.2. Domain decomposition [10 Points]

For the decomposition I went with the most simple cartesian decomposition. As in the last project I used the `MPI_Cart_create` function to create a 2D cartesian grid. The number of processes in each dimension is calculated automatically by MPI. The Domain is the split up evenly in both dimensions for the given number of processes. The code for the decomposition can be found in `code/mini_app/data.cpp`.

2.3. Linear algebra kernels [5 Points]

Most operations are local to the process and can be done sequentially. The only operations that require communication are the dot product and the norm. I implemented the dot product using the `MPI_Allreduce` function and changed the norm to use the dot product. We start by calculating the dot product of the local vectors and then sum them up using `MPI_Allreduce`.

```
for (int i = 0; i < N; i++)
{
    result += x[i] * y[i];
```

```
}

MPI_Allreduce(MPI_IN_PLACE, &result, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

The code for the dot product can be found in `code/mini_app/linalg.cpp`.

2.4. The diffusion stencil: Ghost cells exchange [10 Points]

The only missing part in the implementation here was the exchange of ghost cells. The steps are described here using the north/south direction but are essentially the same for east/west. As outlined in the task description we first need to copy the data from the boundary cells into buffers.

```
for (int i = 0; i <= iend; ++i)
{
    buffN[i] = s_new(i, jend);
    buffS[i] = s_new(i, 0);
}
```

Next we need to send the data to the neighboring processes if they exist. When we set up the cartesian grid we also get the rank of the neighboring processes, if the process has no neighbor the rank is set to a negative value.

```
if (domain.neighbour_north >= 0)
{
    MPI_Isend(&buffN[0], nx, MPI_DOUBLE, neighbour_north, 1, comm_cart, &reqs[cnt++]);
    MPI_Irecv(&bndN[0], nx, MPI_DOUBLE, neighbour_north, 2, comm_cart, &reqs[cnt++]);
}
```

We use this to send and receive the ghost cell data from the neighboring processes (if they exist).

The last step is to wait for all the communication to finish.

```
MPI_Waitall(count, reqs, MPI_STATUSES_IGNORE);
```

2.5. Implement parallel I/O [10 Points]

For this part of the task I started by copying the code from the provided example. The only thing I had to adjust was the storage order, which I set to `MPI_ORDER_FORTRAN`.

2.6. Scaling studies

For both strong and weak scaling benchmarks the code was run for 100 time steps to a final time of 0.005. Each task was run on a separate cpu (`--cpus-per-task=1`) and all tasks on a single node (`--nodes=1`). Additionally all runs were repeated at least 10 times. The generated output files for all runs can be found in `code/mini_app/out`. The benchmarking scripts are in `code/mini_app`. The results are compared to the OpenMP implementation from project03. The full benchmarks for the OpenMP implementation (right in the figures below) can be found in the project03 repository.

2.6.1. Strong scaling [10 Points]

To evaluate the performance of the new implementation I ran the code using $p = 1, 2, 4, 8, 16$ processes and for each process count I ran the code with $s = 64, 128, 256, 512, 1024$ mesh points (resolution) in each direction. The results are shown in Figure 1 and Figure 3.

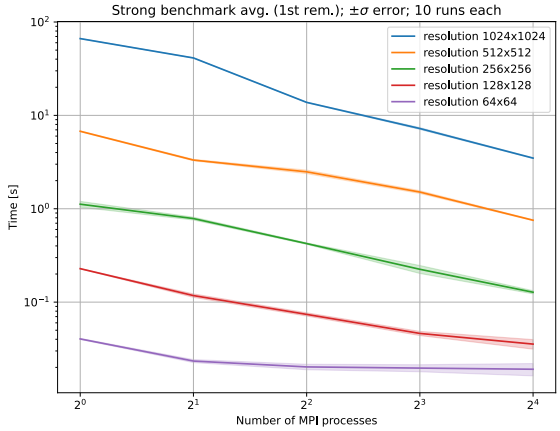

Figure 1: Strong scaling MPI

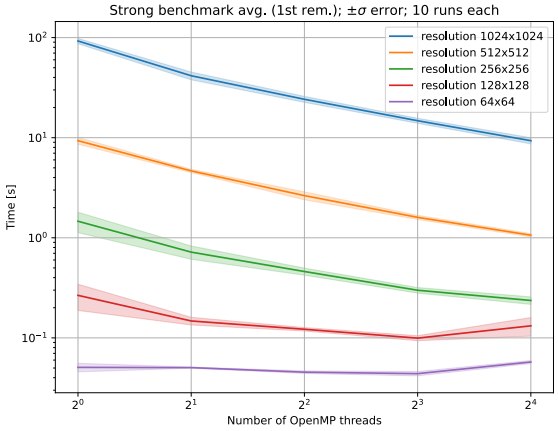

Figure 2: Strong scaling OpenMP

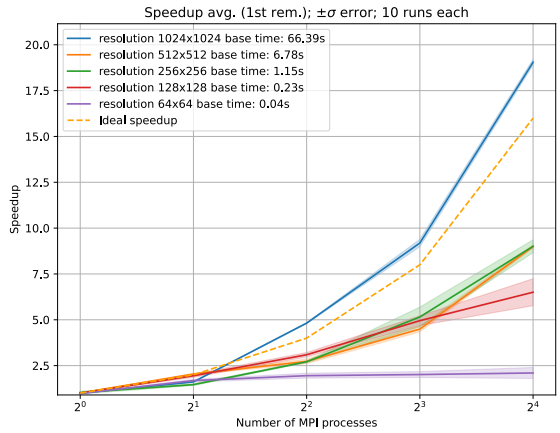

Figure 3: Speedup MPI

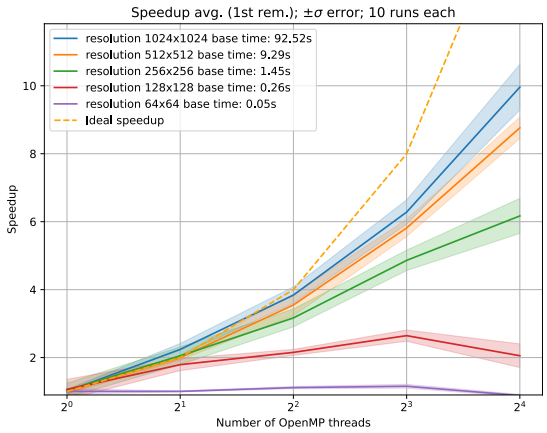

Figure 4: Speedup OpenMP

In Figure 3 we can see that as expected for the smallest mesh size (64) we get the slowest speedup. This is could be caused by the fact that the communication overhead is relatively large compared to the computation time. For the larger mesh sizes the communication overhead is less significant and we get a better speedup. We can also see for the largest mesh size (1024) we get a super-linear speedup. One possible explanation for this is that the larger mesh we exceed the cache size and the communication overhead becomes less significant. A quick calculation shows that the AMD EPYC 7763 has a L2 cache of 512KiB per core which would allow for a mesh size of about $\sqrt{\frac{512\text{ KiB}}{8}} \approx 256$ mesh points in each direction to fit into the L2 cache.

We can see that both the MPI and OpenMP implementations scale similarly. The MPI implementation is consistently faster, especially for the largest mesh resolution (1024).

2.6.2. Weak scaling [10 Points]

For the weak scaling benchmark I ran the code using $p = 1, 4, 16, 64$ processes and a base sizes of $s = 64, 128, 256$ mesh points in each direction. The number of mesh points for each configuration was calculated as:

$$\#\text{ mesh points} = \left\lfloor \text{base\_size} * \sqrt{\#\text{ processes}} \right\rfloor^2$$

The results are shown in Figure 5 and Figure 7.

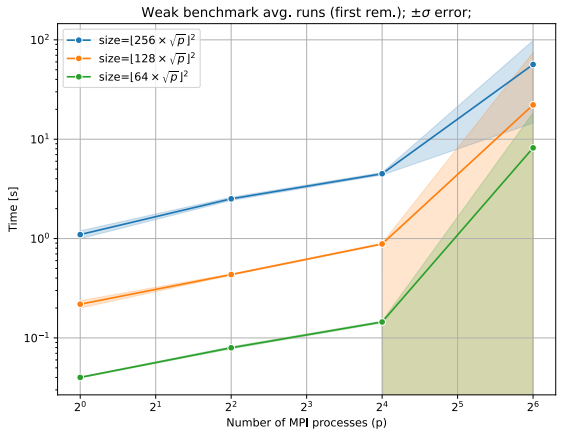

Figure 5: Weak scaling MPI

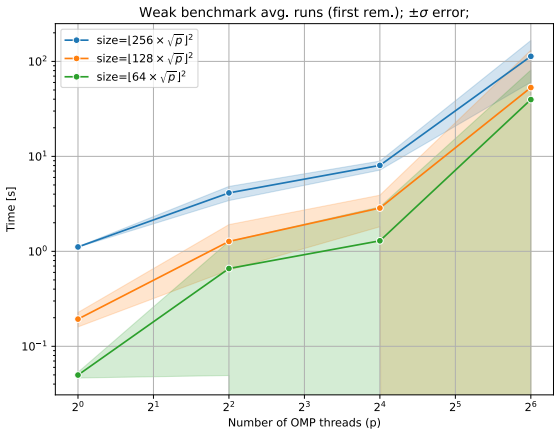

Figure 6: Weak scaling OpenMP

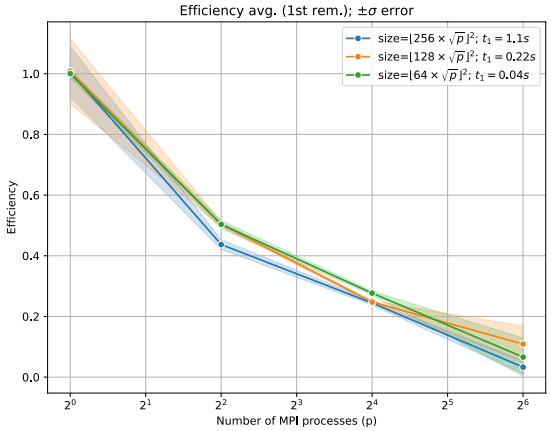

Figure 7: Efficiency MPI

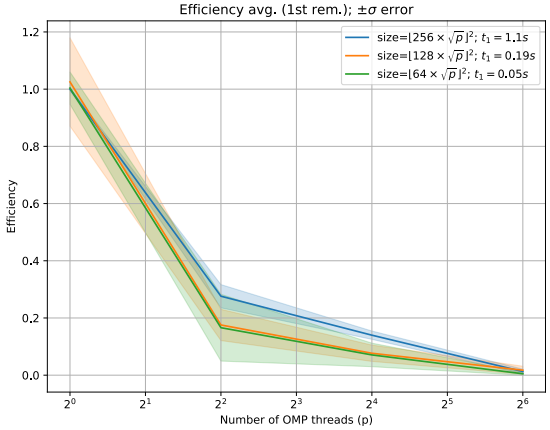

Figure 8: Efficiency OpenMP

Figure 5 shows us that all three tested base sizes scale very similarly. Only for 64 processes and the smallest base size we see a significant drop in performance. This is likely due to the communication overhead which becomes more significant for a larger number of processes. In Figure 7 we see an initial drop in efficiency which is likely due to the communication overhead when we switch from a single process to multiple. When we increase the number of processes the efficiency decreases more slowly. However for 64 processes the efficiency drops below 20% for all base sizes.

Again comparing Figure 7 and Figure 8 we can see that they scale similarly. The MPI implementation is a little bit faster and we can notice a more significant initial drop in efficiency for the OpenMP implementation.

To achieve the best performance we should probably implement a hybrid strategy. Parallelize the code on one node using OpenMP and share the work among the nodes using MPI.

3. Python for High-Performance Computing [in total 40 points]

3.1. Sum of ranks: MPI collectives [5 Points]

For this task I translated the cpp code from project04/ring to python. As outlined in the task description I implemented one version which communicates using python objects and another version which uses `NumPy` arrays. To test the code I ran the following commands:

```
mpirun -np 8 python3 slow_comm.py | sort > slow_comm.txt
mpirun -np 8 python3 fast_comm.py | sort > fast_comm.txt
```

The respective code and text files with the output can be found in `code/hpc_python/rank_sum`.

3.2. Ghost cell exchange between neighboring processes [5 Points]

Again i started this task by translating cpp code this time from project04/ghost to python. The implementation of the ghost cell exchange using `NumPy` arrays was pretty straight forward except for the sending of the first and last columns. As far as I could tell `mpi4py` doesn't really support sending non memory contiguous data. To work around this I created a copy of the data I wanted to send as a contiguous array.

```
left_s = data[1, 1:-1].copy()
right_s = data[-2, 1:-1].copy()
```

We also need to create a contiguous receiving buffer for the ghost cells.

```
left_r = np.zeros(SUBDOMAIN, dtype=np.int64)
right_r = np.zeros(SUBDOMAIN, dtype=np.int64)
```

After sending the ghost cells we can receive them and copy them into the correct position.

```
data[1:-1, 0] = left_r
data[1:-1, -1] = right_r
```

To test the code I ran the following commands:

```
mpirun -np 16 python3 ghost.py
```

Which as expected prints the correct output:

```
[[ 9  5  5  5  5  5  5  9]
 [ 8  9  9  9  9  9  9 10]
 [ 8  9  9  9  9  9  9 10]
 [ 8  9  9  9  9  9  9 10]
 [ 8  9  9  9  9  9  9 10]
 [ 8  9  9  9  9  9  9 10]
 [ 8  9  9  9  9  9  9 10]
 [ 9 13 13 13 13 13 13  9]]
```

3.3. A self-scheduling example: Parallel Mandelbrot [30 Points]

For this task we were provided with a single processes implementation for calculating the Mandelbrot set. As suggested in the provided template I implemented two key functions: one for the master process and one for the worker processes. The master process is responsible for distributing the work to the worker processes and the worker processes are responsible for calculating the Mandelbrot set for a given patch.

The basic steps are:

1. The master process calculates the patches and sends them to the worker processes. (implementation provided)
2. The master process sends an initial patch to each worker process.
3. The worker processes calculate the Mandelbrot set for the given patch.
4. The master process waits for a worker process to finish and sends a new patch to the worker process if there are any left
5. Back to step 3 until all patches are calculated.
6. The master process sends a message to the worker processes to stop and assembles the final image.

The implementations are in `code/hpc_python/ManagerWorker`.

To test evaluate the performance of the parallel Mandelbrot implementation I ran the code using $p = \{2..32\}$ processes and $t = \{50, 100\}$ patches. The results are shown in Figure 9 and Figure 10.

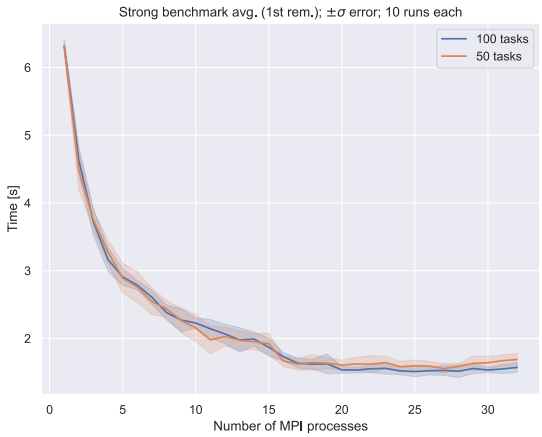

Figure 9: Strong scaling

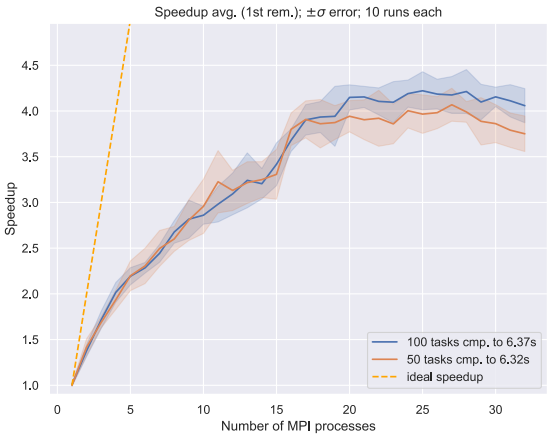

Figure 10: Speedup

We can see that initially we get a decent speedup but adding more than about 10 processes doesn't really improve the performance much anymore. This is likely due to the fact that the patches are relatively small and the communication overhead becomes significant. The difference between 50 and 100 patches is not large and only really evident for $p > 20$ processes.