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. Not that much to say here.

2.2. Domain decomposition [10 Points]

2.3. Linear algebra kernels [5 Points]

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

2.5. Implement parallel I/O [10 Points]

2.6. Strong scaling [10 Points]

2.7. Weak scaling [10 Points]

2.8. Bonus [20 Points]: Overlapping computation/computation details

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 1 and Figure 2.

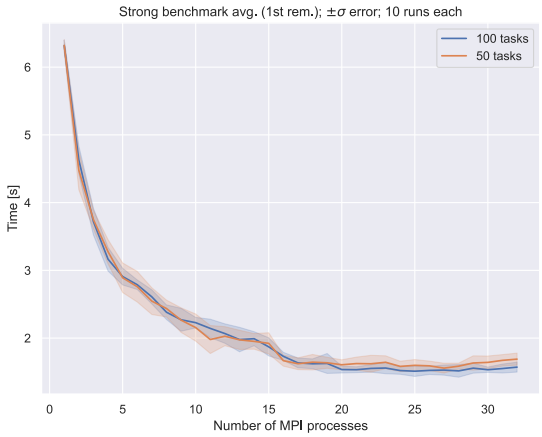

Figure 1: Strong scaling

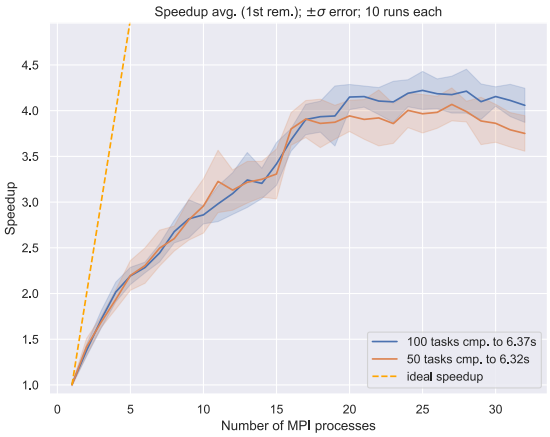

Figure 2: Speedup

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.