ETH zürich

High-Performance Computing Lab for CSE **2020**

Student: Konrad Handrick Discussed with: -

Solution for Project 6

Due date: May 31, 2020, 12pm (midnight)

Disclaimer for all exercises: I benchmarked most of my programs but when I was trying to collect plots and data from the login node, the attack happened and as of now (May 31st), I cannot access these data anymore, that's why I present only benchmarks from my i7 8th gen quadcore machine.

Everything runs smoothly on my local machine and should work as expected loading the right modules from Euler.

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 40 points]

For this exercise, I really mostly followed the instructions that were given in the comments of the code.

1.1. Initialize and finalize MPI [5 Points]

Nothing to add. Code is straightforward.

1.2. Create a Cartesian topology [10 Points]

Here again we need to adapt to MPI's internal ordering of topologies. Again, more or less straightforward following the template's comments on what to do.

1.3. Extend the linear algebra functions [5 Points]

Here we too follow the code's comments. I tried enhancing performance to a maximum but unfortunately, MPI and Intel Intrinsics do not seem to work well together resulting in a Seg Fault when pointers to intrinsics functions in an MPI program. I suspected non-contiguous memory alignment on receiving ranks such that the rank tries to access memory which is out-of-bounds. The MPI standard by default does allocate contiguous memory so I'm at a loss where to look for the error here.

The code I now commented out works perfectly well in a sequential program but not in an MPI-enabled one. Nevertheless did loop unrolling to enhance performance by a small margin which I left in for both the functions that had to be implemented. Applying collective operations involved checking the documentation and not much else.

1.4. Exchange ghost cells [10 Points]

I tried interleaving communication and computation as suggested in the project's PDF. The ranks need to perform a synchronize operation (BARRIER) though. The non-blocking point-to-point operations seem to work well.

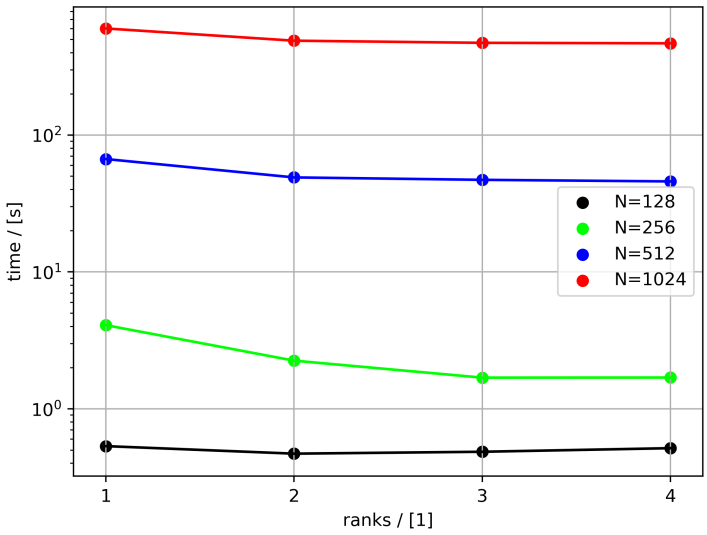

Figure 1: Scaling of the PDE-Miniapp

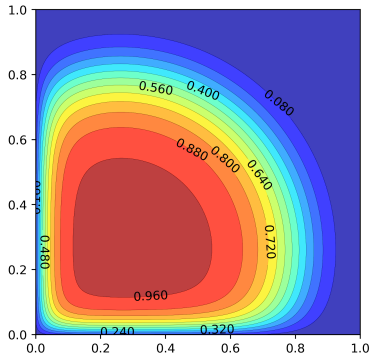

Figure 2: Result of the stencil computation

1.5. Scaling experiments [10 Points]

I was very happy to observe nearly superlinear scaling on the Euler nodes. Unfortunately, my local machine does not seem to exhibit the same behavior. A speedup is most visible at medium ($N = 256$ up to $N = 768$) problem sizes where the program shows speedup by up to 3.5 times.

2. Python for High-Performance Computing (HPC) [in total 60 points]

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

I coded a file called "sum_ranks.py" which you'll need to launch using Python 3. Passing the argument "pickle" will yield the pickle-based (serialized) point-to-point communications of mpi4py whereas the argument "buf" will yield the buffered, apparently faster version. I opted for a ring-buffer implementation.

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

To solve this one, I wrote the file "cart_decomp.py" which uses pickle-based functions and outputs a sanity check.

2.3. Exchange rank with neighbours [5 Points]

To have a ghost cell exchange, I made use of a cartesian decomposition and buffered methods. A sanity check output is provided. The corresponding file is name "ghost_exchange.py".

2.4. Change linear algebra functions [5 Points]

Here, I followed the basic PDE-miniapp implementation which we did in the first exercise and used buffered methods.

2.5. Exchange ghost cells [5 Points]

Implementing a good "wait" procedure took me some time. Else, this just followed from documentation and again, from the first exercise.

2.6. Scaling experiments [5 Points]

While on Euler I saw some performance, on my local machine the program exhibits more degrading performance the more ranks are used. This seems to come from the overhead introduced by mpi4py, ie making the intermediate representation of the code very slow.

Looking at the scaling plots we can conclude that the Python version, in any case, is by order(s) of magnitude slower than the C version but has the nice effect of not needing to instantiate as much boiler plate code. Either you use it as purely prototyping some ideas or you interface Python in a different way - clearly, the code generated by mpi4py is not competetive in an HPC context.

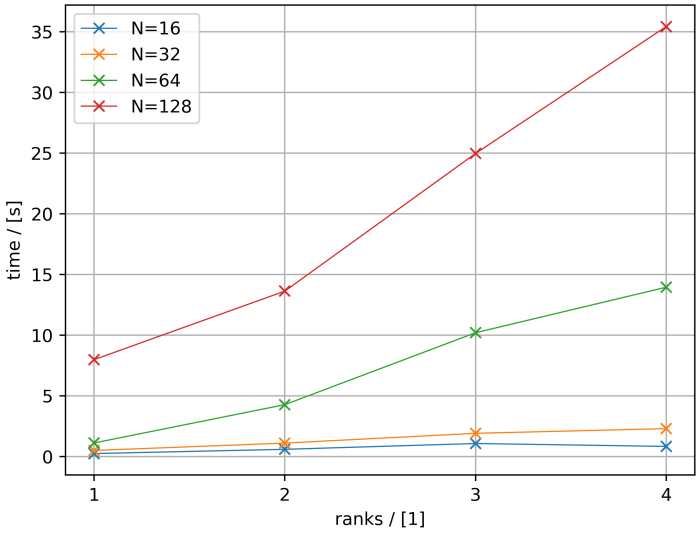

Figure 3: Scaling of the PDE-Miniapp, Python version

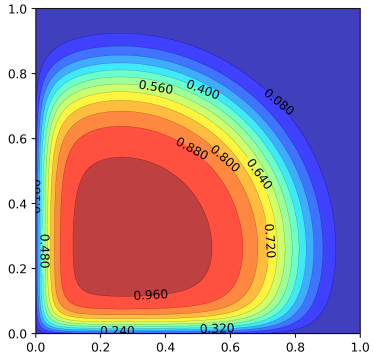

Figure 4: Benchmarking the ManagerWorker example on different size grids and varying number of tasks

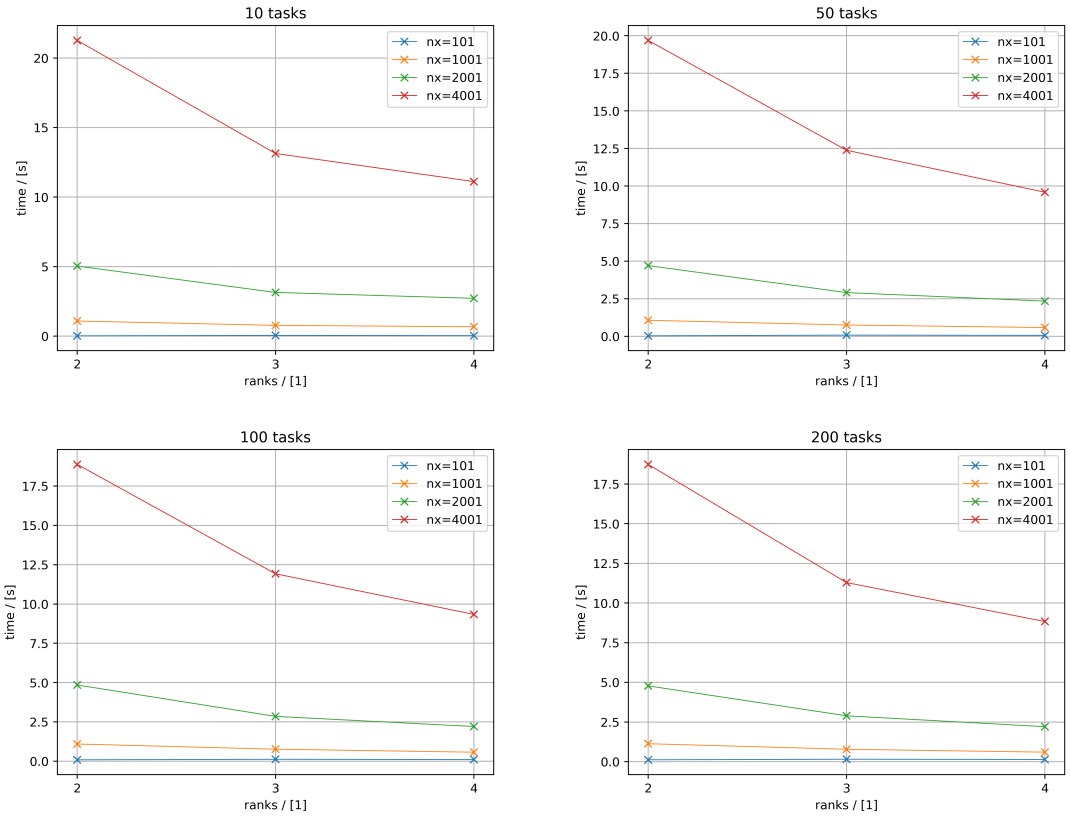

Figure 5: Result of the stencil computation, Python version

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

Interestingly in a producer-consumer context pythonic MPI started to shine. Be it on Euler or my local machine, the parallelization scales well and works as intended. To solve this exercise, I left in the proposed 1d decomposition of the grid and did not pursue more involved decompositions.

The master rank needs to kick off the calculation and then iteratively distribute the tasks among the ranks, which in theory should have the benefit of using the hardware more efficiently since no rank should wait for a too long time if done well.

Collecting the data shows that the overhead generated by the master rank is rather low ($< 1\%$ load imbalance, see the raw data file).

I ran a few diagnostics which showed that this paradigm works well (in the above mentioned efficiency sense), mpi4py in a producer consumer context generates not too many instructions and the hardware is used rather efficiently also.

For the first diagnostic I compared load imbalances which I won't put up here since the values only differ marginally - essentially, the master rank had about 101 and 103 percent of the computation time compared to the slave ranks.

For the second metric I generated a flame graph proposed by Brendan Gregg which visualizes a stack trace. We can observe on the left hand side that on startup, python generates a lot of work which is subsequently loaded by the processors in a rather efficient manner - you can barely record the distribution of the work.

Lastly, perf showed that caches are used efficiently and not many ineffiencies happen on a lower level (ie not a lot of branhc misses or extensive numbers of context switches).

Finally, the time-to-solution varies a lot with how many tasks are distributed among the ranks. We can deduce that with more ranks and large enough problem size, more tasks distribute much more efficiently than the other way round.

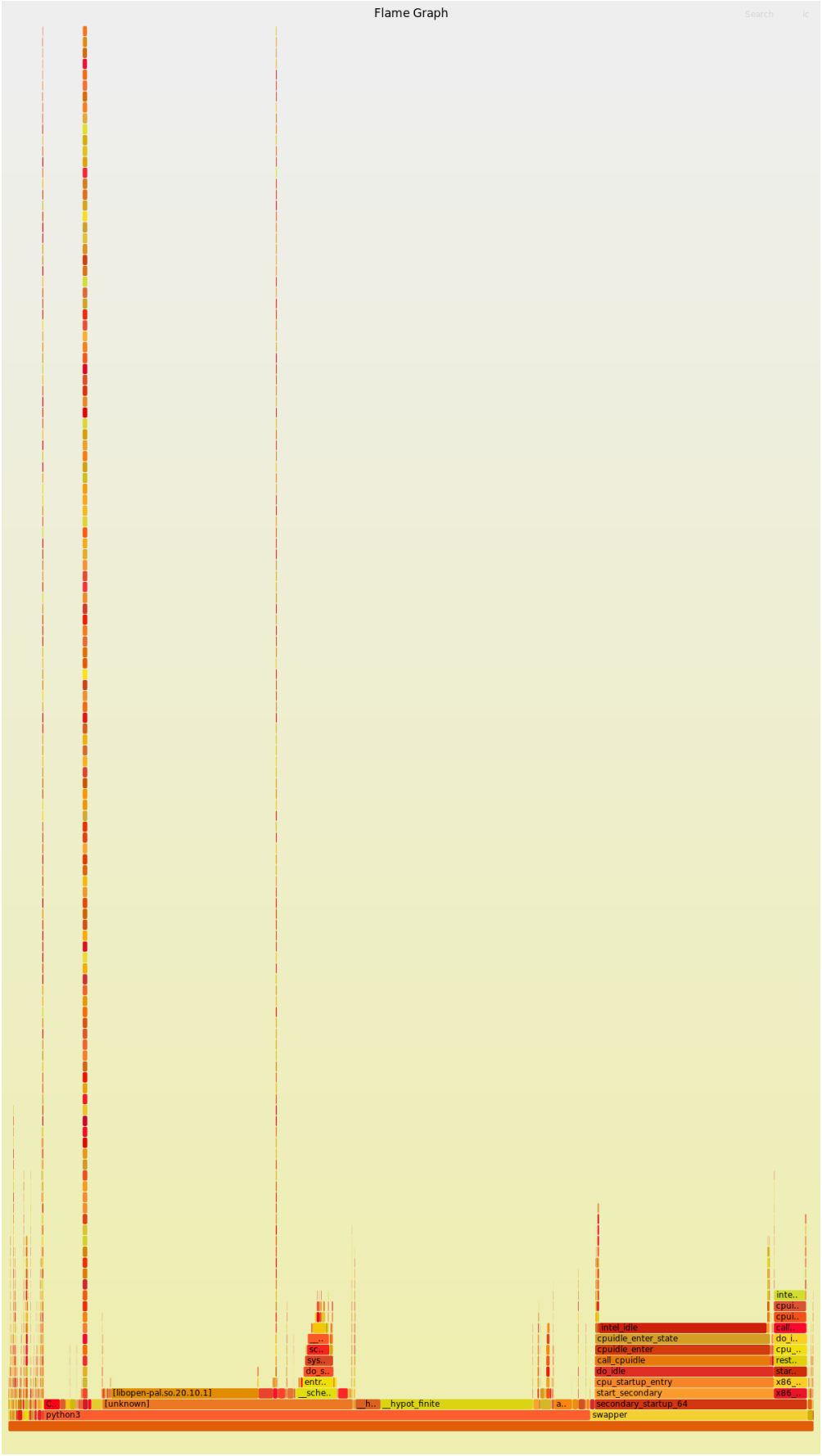

Figure 6: Flame Graph of the stack of the Manager-Worker example using 4 ranks

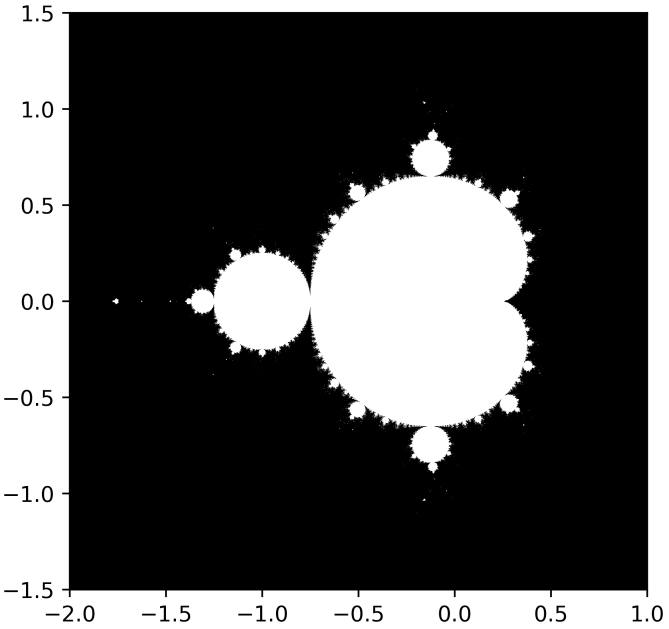

Figure 7: Mandelbrot solution generated on a 4001x4001 grid using 4 ranks

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 observations for all exercises by writing an extended Latex report. Use the Latex template provided on the webpage and upload the Latex summary as a PDF to Moodle.

- 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 MPI solutions;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .zip/.tgz through Moodle.