University of Basel

Lecturer:
Prof. Dr. **Florina M. Ciorba**
Prof. Dr. Erick Lavoie
Prof. Dr. Heiko Schuldt
Prof. Dr. Isabel Wagner

Seminarraum 05.002,
Spiegelgasse 5,
CH-4051 Basel

Assistants:
Ali Ajorian, M.Sc
Jonas Korndörfer, M.Sc
Shiva Parsarad, M.Sc
Yiming Wu, M.Sc

https://vorlesungsverzeichnis.unibas.ch/de/home?id=277318

Foundations of Distributed Systems 2023 (45402-01, 45402-02) **Fall Semester 2023**

Assignment 6 - Hybrid OpenMP + MPI and Performance *(50 Points)*

Starting Date: November 07, 2023
Deadline: November 20, 2023 - 23:55

Assistant contact:
jonas.korndorfer@gmail.com

The first thing you need to do is **register your group in the Google sheet** provided by the following link:
Google sheet to register your group
This sheet will also be used for setting the time slots for the assignments evaluation.

In this assignment you will work on the miniHPC cluster (see attached description **miniHPC_Usage.pdf**).
To connect to the cluster use:
ssh -X UserName@cl-login.dmi.unibas.ch
The **UserName** is the same as your short name for the university of Basel and the **password** is the same of your UniBas email.

The delivered solutions should be in a single **tar file.**
To create the **tar file** use the following command:
```
tar -zcvf archive-name.tar.gz directory-with-your-solutions
```

You can find a lot of information, definitions and syntax for OpenMP and MPI commands in the course slides. Furthermore, you can also consult the following URLs:

- OpenMP
- OpenMP 5.0 Complete Specifications
- An open source MPI implementation

To execute and test your solution for this exercise, create a job script similar to the following example. This is a commented (a line prepended by #) example to help you understand what each variable in the job script represents. The example below is also attached in a separate file (JobScriptExample.job) to the exercise. To execute a job script on miniHPC, use **sbatch your-job-script-file-name.job**

```
#!/bin/bash

#SBATCH -J TEST # Job name.
#SBATCH --time=00:05:00 # Maximum estimated running time for this job.
#SBATCH --exclusive # You will run experiments exclusively on the allocated nodes.
#SBATCH --nodes=2 # Number of nodes you request from the batch scheduler.
#SBATCH --ntasks-per-node=2 # Number of MPI ranks per node. If you use nodes=2 and ntasks-per-node=2 you will be running 4 ranks.
#SBATCH --cpus-per-task=2 # Number of OpenMP threads.
#SBATCH --partition=xeon # Which node partition you request nodes from. You can find more information about the partitions on the attached miniHPC_Usage.pdf file.
#SBATCH --output=OUTPUT.txt # The files where the output of your program will be directed to.
#SBATCH --hint=nomultithread # Disable the usage of hyperthreads.

ml intel # Load the intel compiler.
srun yourCode.o 10000 1024 0 0 0.75 # Run your code for this exercise with the needed parameters.
```

Recall that the Xeon nodes on miniHPC have only 20 cores each and that you will use 8 OpenMP threads per rank in certain experiments. Therefore, you can not place more then 2 MPI ranks per node. For the parallel experiments use a maximum of `ntasks-per-node=2` this way you ensure that you will not have more than 2 ranks on the same node.

Please make sure that any optimization you introduce does not affect the correctness of the results.

1 Task - Hybrid MPI + OpenMP Programming: (25 points)

Optimize and parallelize the provided sequential Mandelbrot code `mandel_seq.c`. The parallelization should use MPI and OpenMP. You will need to calculate and divide the loop iterations (work) to each MPI rank. Then, you will also need to parallelize using OpenMP the processing (function `calculate_pixel` of the code) that each rank will execute. You will then be explicitly taking advantage of two levels of parallelism: the scalability of distributed memory and the speed of shared-memory.

Remember that to compile a **hybrid MPI + OpenMP** program you will need to write the following command in your job script:

```
module load intel
mpiicc -O3 -fopenmp hello.c -o hello
```

The provided code skeleton must be executed with the following command in your job script:

```
srun mandel_seq.o 10000 1024 0 0 0.75
```

The parameters represent the number of iterations, the size, the x0, the y0, and the zoom, respectively. All of these parameters affect the results and execution time of the Mandelbrot program and you can play with them. However, for the experiments in this exercise you should use the parameter values listed above.

The Mandelbrot code will generate a CSV file. The information in this file can be visualized with the provided Python script (`visualize.py`). To run that script execute the following command:

```
python visualize.py
```

- Deliverables:
  - The hybrid MPI + OpenMP source code. *(25 Points)*
    If you deliver the hybrid version you do not need to provide the following codes.
  - The source code with your parallel version using OpenMP. *(7 Points)*
  - The source code with your parallel version using MPI. *(18 Points)*

2 Task - Performance Measurement: (25 points)

This task consists in the evaluation of the performance of your parallel version of the Mandelbrot code.

Before you start the task, recall the following information from performance analysis:

Speedup can be translated as the gain made in speed by parallel vs. sequential execution. Therefore, speedup S_P is the ratio of the time taken to solve a problem on a single processor T_S to the time required to solve the same problem on a parallel T_P computer with identical processing elements PE.

$S_P = \frac{T_S}{T_P}$

Efficiency E can be translated as how efficient a parallel program is, considering the resources used. It is the ratio between the performance achieved (the speedup) and the resources used to achieve that performance. For example, if you paralellize a program for 2 PEs you expect that the program will run twice as fast as the sequential version in a perfect system. Therefore, the efficiency of a parallel program can be calculated by:

$E = \frac{S_P}{PE}$

Recall that in the case of a hybrid code the resources are mixed. Therewith, if you have 2 threads for each rank and you have 2 ranks, you will end up with 4 PEs.

You will need to generate certain speedup and efficiency graphs.
For the experiments you must repeat the execution of your code 5 times and use the average of the execution times.

- Hand in the following speedup graphs for your code.
  - Speedup graph for your parallel version increasing only the number of OpenMP threads (sequential*, 2, 4, 8). Five executions for each configuration. *(4 Points)*
  - Speedup graph for your parallel version increasing only the number of MPI ranks (sequential*, 2, 4, 8). Five executions for each configuration. *(4 Points)*
  - Speedup graph for your parallel version increasing both the number of MPI ranks and OpenMP threads (sequential*, 2, 4, 8). Five executions for each configuration. *(5 Points)*
- Hand in the following efficiency graphs for your code.
  - Efficiency graph for your parallel version increasing only the number of OpenMP threads (sequential*, 2, 4, 8). Five executions for each configuration. *(3 Points)*
  - Efficiency graph for your parallel version increasing only the number of MPI ranks (sequential*, 2, 4, 8). Five executions for each configuration. *(4 Points)*
  - Efficiency graph for your parallel version increasing both the number of MPI ranks and OpenMP threads (sequential*, 2, 4, 8). Five executions for each configuration. *(5 Points)*

*** You can reuse the times collected for the 5 sequential executions for all experiments.**