University of Basel

Lecturer:
Prof. Dr. **Florina M. Ciorba**
Prof. Dr. Heiko Schuldt
Prof. Dr. Christian Tschudin
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/en/home?id=285227

Foundations of Distributed Systems 2024 (45402-01, 45402-02) **Fall Semester 2024**

Assignment 4 - Hybrid OpenMP + MPI and Performance *(20 Points)*

Starting Date: October 22, 2024
Deadline: November 21, 2024 - 23:55

Assistant contact:
jonas.korndorfer@unibas.ch

Useful information

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 registration and sheet will be used for setting the time slots for the assignments interview & 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`, where `UserName` is the short name of your UniBas account and `password` is your UniBas password.

To login to miniHPC you must be connected to the UniBas VPN (instructions available online[1]).

The solutions of this assignment must be delivered in a zip or tar file containing:

- All source codes containing the solutions.
- All job scripts used to execute and collect performance measurements of the exercises.
- A single PDF file containing the requested plots and or written answers.

You can find rich information, definitions, and syntax for OpenMP and MPI calls in the lecture slides. You can additionally consult the following URLs:

- OpenMP https://www.openmp.org
- OpenMP 5.2 Complete Specifications https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf
- OpenMPI https://www.open-mpi.org/

[1] https://its.unibas.ch/de/anleitungen/netzwerkzugang/anleitung-vpn/

To execute and test your solution for this exercise, create a job script similar to the following example. In this example, lines prepended by #SBATCH inform the batch scheduler about the desired job configuration. The second # on each line denotes a comment, to help understand what each variable in the job script represents. This example is also included in a separate file (JobScriptExample.job) in the exercise handout. 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. With nodes=2 and ntasks-per-node=2 you will be running on 4 MPI ranks.
#SBATCH --cpus-per-task=2 # Number of OpenMP threads.
#SBATCH --partition=xeon # Partition from which you request nodes. More information about the partitions can be found in the attached miniHPC_Usage.pdf file.
#SBATCH --output=OUTPUT.txt # File where the output of your program will be directed to.
#SBATCH --hint=nomultithread # Disable the usage of hardware multithreading.

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

Recall 1. 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`** to ensure no more than 2 MPI ranks on the same node.

Recall 2. To compile an **MPI + OpenMP** program you should use the following command:

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

Please ensure and verify that any optimization you introduce does not affect the correctness of the results.

1 Task: MPI + OpenMP General Matrix Multiplication (GEMM) (4 points)

GEMM remains a fundamental operation in numerous high performance computing applications, such as machine learning, scientific simulations, and graphics rendering. Its optimization is critical for improving the performance of applications in these domains. Despite its long history, GEMM is continuously refined and adapted to modern computing systems. A recent publication "Comparing the Performance of General Matrix Multiplication Routines on Heterogeneous Computing Systems"[2] of the Journal of Parallel and Distributed Computing (2022) highlights recent advancements in optimizing GEMM for heterogeneous platforms, showcasing its ongoing relevance in cutting-edge research.

In this exercise you will parallelize GEMM using MPI and OpenMP.
You will start with the provided `codeSkeletons/gemm.c` code skeleton, which contains several TODOs. Follow the TODOs and introduce the necessary code to efficiently parallelize *gemm.c* using MPI and OpenMP.

a. Distribute the workload such that each rank computes a portion of the GEMM. *(1 Point)*

b. Parallelize the local GEMM in each MPI rank using OpenMP. *(1 Point)*

c. Execute the parallelized code using matrix size 4000, 2 ranks, and 10 OpenMP threads per rank. *(1 Point)*

d. Execute the parallelized code using matrix size 4000, 4 ranks, and 10 OpenMP threads per rank. *(1 Point)*

Hand in the source code and the results from the executions.

[2] https://www.sciencedirect.com/science/article/pii/S0743731521001933

2 Task: MPI + OpenMP Mandelbrot (6 points)

Optimize and parallelize the provided sequential Mandelbrot code `mandel_seq.c`. The parallelization should use MPI and OpenMP. You will need to divide the loop iterations (data) to each MPI rank. Then, you will need to parallelize using OpenMP the work (function `calculate_pixel` in the code) that each MPI rank will execute. This will allow to explicitly take advantage of two-level parallelism: the scalability of distributed memory for cross-node parallelism and the speed of shared-memory for node-level parallelism.

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 (10000), the size (1024), the x0 (0), the y0 (0), and the zoom region (0.75), respectively. All these parameters affect the result and execution time of the Mandelbrot program and are, therefore, something you can play with to see these differences. However, for the experiments in this exercise you must 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`). Run the script with: **`python visualize.py`**.

Deliverables:

a. The hybrid MPI + OpenMP source code. *(6 Points)*
If you deliver the hybrid version you do not need to provide the codes of deliverables b and c.

b. The source code with your parallel version using OpenMP. *(2 Points)*

c. The source code with your parallel version using MPI. *(4 Points)*

3 Task: Performance Measurement: (10 points)

This task consists in the evaluation of the performance of your parallel versions of codes from the first two tasks: GEMM and Mandelbrot.

For this task, recall the following about performance analysis:

Speedup S_P is the gain made in speed by parallel vs. sequential execution. Therefore, speedup 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 describes how efficiently a parallel program uses the resources. It is calculated as the ratio between the achieved speedup and the resources used to achieve that performance. For example, if execute a parallel program on 2 PEs you expect that it runs twice as fast as the sequential version. 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. This means, if you have 2 threads for each rank and you have 2 ranks, you will actually have 4 PEs.

You will need to generate speedup and efficiency graphs. **Repeat the execution of your code 5 times (for each configuration) and report the average, min and max of their execution times.**
Deliverables:

- Hand in the following **speedup** graphs:
  a. Speedup of your parallel version increasing only the number of OpenMP threads (sequential*, 2, 4, 8).
  b. Speedup of your parallel version increasing only the number of MPI ranks (sequential*, 2, 4, 8).
  c. Speedup of your parallel version increasing both the number of MPI ranks and OpenMP threads (sequential*, 2, 4, 8).
- Hand in the following **efficiency** graphs:
  a. Efficiency of your parallel version increasing only the number of OpenMP threads (sequential*, 2, 4, 8).
  b. Efficiency of your parallel version increasing only the number of MPI ranks (sequential*, 2, 4, 8).
  c. Efficiency of your parallel version increasing both the number of MPI ranks and OpenMP threads (sequential*, 2, 4, 8).

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