Lecturers
Prof. Dr. **Florina M. Ciorba**
Prof. Dr. Heiko Schuldt
Prof. Dr. Christian Tschudin
Prof. Dr. Isabel Wagner

Assistants
Ali Ajorian
Thomas Jakobsche
Valentyna Pavliv
Yiming Wu

Info
https://adam.unibas.ch/go/crs/1995845
Seminarraum 05.002
Spiegelgasse 5, Basel

Exercise 5: Intro to miniHPC and OpenMP (20 points)

Given: November 06, 2025
Deadline: November 16, 2025 (23:55)
Contact: thomas.jakobsche@unibas.ch

Instructions

- All exercise regulations discussed in the lecture and previous exercises apply.
- You will complete this exercise by executing programs on the miniHPC cluster.
- This exercise must be completed in groups of four and handed in by one.
- Submissions are made through ADAM and will be followed by interviews.
- Hand-in all files (e.g. scripts, codes, outputs) and a PDF with answers.

Objectives

- Hands-on work using an HPC system with a resource manager.
- Learn how to program, compile, and execute OpenMP[1] code in C.
- Understand data parallelism (work sharing loops) and memory architecture.

Tasks

- Task 1: Connecting to miniHPC and using Slurm (2 points)
- Task 2: Investigating Memory Architecture (4 points)
- Task 3: "Hello World" OpenMP and Job Configuration (2 points)
- Task 4: Data Parallelism through Work Sharing Loops (8 points)
- Task 5: Scaling OpenMP and Data Access Patterns (4 points)

How to use miniHPC

- You will work on miniHPC, a computing cluster for research and teaching purposes (see more https://hpc.dmi.unibas.ch/research/minihpc/).
- Login: `ssh <shortname>@cl-login.dmi.unibas.ch` (UniBas short name and password, you need to be connected to the UniBas VPN). After logging in, you will be at your home directory on the miniHPC login node.
- The login node is the starting point to interact with the cluster, you should not execute your programs on the login node. Use the Slurm[2] resource manager to

[1] https://www.openmp.org
[2] https://slurm.schedmd.com

submit computing "jobs" to the multiple nodes of the cluster.
- The Slurm resource manager coordinates your (and others) jobs and handles the allocation and scheduling of jobs onto nodes. Your jobs may wait in the job queue if nodes are busy with other jobs.
- To move files between your local machine and the miniHPC cluster, you can use the `scp` command and appropriate flags (e.g., `-r` to copy directories recursively).
- To run a program on compute nodes, request resources with a job script. Submit it using `sbatch <job_script>`, cancel jobs with `scancel <job_id>`, and check their status using `squeue`. Job output is saved in `slurm-<job_id>.out`.
- **It is OK to compile your programs on the login node, but do not execute them on the login node, always submit them as jobs.**

OpenMP instructions

- You can compile your OpenMP C program with GCC[a] by first loading the newest installed GCC compiler with `module load GCC/14.2.0` and then compiling with `gcc -O3 -fopenmp my_program.c -o my_program`.
- Set the number of OpenMP threads to the requested number of threads inside your job script with `export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}`.

[a] https://gcc.gnu.org

Task 1: Connecting to miniHPC and using Slurm (2 points)

Connect to miniHPC and learn how to submit a job with the Slurm resource manager.

1. Connect to miniHPC with `ssh` following the "How to use miniHPC" instructions.
2. Submit a simple job script with `sbatch` to return the `hostname` of your allocated compute node. (*Hint:* You can use the example job script shown in Listing 1.)
3. Report the output of your Slurm job. Which node was allocated to your job?

```
[wang0041@dmi-cl-login ~]$ sbatch jobscript
Submitted batch job 2454288
[wang0041@dmi-cl-login ~]$ cat slurm-2454288.out
dmi-cl-node003.dmi.p.unibas.ch
```

dmi-cl-node003.dmi.p.unibas.ch

Task 2: Investigating Memory Architecture (4 points)

Understand and report the memory architecture of miniHPC.

1. Investigate the memory architecture of miniHPC `xeon` nodes (e.g. the login node). (*Hint:* Use the commands `sinfo -p xeon -N`, `lscpu`, and `lstopo --no-io --of txt` you can also read more about miniHPC here https://hpc.dmi.unibas.ch/research/minihpc/. If you want to execute `lstopo` on a compute node you need to first load the appropriate module with `module load hwloc` before you can use the command.)
2. Report miniHPC memory architecture in lecture terms. What kind of system is it?
3. What is a NUMA domain? How many NUMA domains does one `xeon` node have?

Task 3: "Hello World" OpenMP and Job Configuration (2 points)

Write a C program that prints "Hello World from thread ID: <thread ID>" with multiple OpenMP threads. Write a job script to execute your program on miniHPC (see an example job script in Listing 1).

1. Record OpenMP runtime information such as `omp_get_num_procs()`, `omp_get_num_threads()`, and `omp_get_max_threads()` in your code.
2. Submit four jobs with different combinations of `ntasks-per-node` $\in \{1, 2\}$ and `cpus-per-task` $\in \{1, 2\}$. Record how many *processes* are launched and how many *threads* each process uses.
3. Explain the resulting behavior and why output lines are ordered (or not).

Task 4: Data Parallelism through Work Sharing Loops (8 points)

Parallelize the program in Listing 2 using **8 OpenMP threads** and different approaches. Record, compare, and discuss execution times.

1. Execute the sequential version of the program and record execution time.
2. Parallelize the loops with a manual SPMD approach using only one `#pragma omp parallel` region with thread-local partial sums indexed by `omp_get_thread_num()`, sum up partial sums afterwards. Record execution time.
3. Replace the manual thread-indexing for the loops with multiple `#pragma omp for` and use the OpenMP `reduction` clause where appropriate to avoid having to manually handle partial sums. Record execution time.
   a) Add the `nowait` clause where appropriate and record execution time. Explain why the execution time is (or is not) changing.
4. Compare and discuss the execution times of your different implementations (sequential, manual SPMD, `#pragma omp for`, and `#pragma omp for` + `nowait`).

Task 5: Scaling OpenMP and Data Access Patterns (4 points)

Execute any parallelized version of the program in Listing 2 with different thread count configurations, record execution times, and discuss implications.

1. Execute your parallelized program with `cpus-per-task` $\in \{1, 2, 4, 8, 16\}$ and record execution times. (*Hint:* You can also pass the number of threads through the `sbatch` command with e.g. `sbatch --cpus-per-task=<#threads> jobscript`.)
2. Discuss the implications of execution times for different thread counts. Does the execution time scale proportionally to the number of threads, why or why not?

Listing 1: Example Job Script.

```
#!/bin/bash
#SBATCH --job-name=my_job     # The name of the job
#SBATCH --partition=xeon      # The main partition of miniHPC
#SBATCH --hint=nomultithread # Disable hardware multithreading
#SBATCH --exclusive           # Exclusive access to resources
#SBATCH --time=00:10:00       # Requested execution time
#SBATCH --nodes=1             # Requested number of nodes
#SBATCH --ntasks-per-node=1   # Requested processes per node
#SBATCH --cpus-per-task=1     # Requested threads per process

# Tell Slurm to execute something (e.g. your code)
# We execute the hostname command as a placeholder
srun hostname
```

Listing 2: Sequential C code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void) {
    const size_t n = 1000000000;
    double *data = malloc(n * sizeof *data);
    if (!data) { return 1; }
    double start = omp_get_wtime();

    // LOOP 1: create synthetic data
    for (size_t i = 0; i < n; ++i) { data[i] = i % 10; }

    // LOOP 2: compute sum
    double sum = 0.0;
    for (size_t i = 0; i < n; ++i) { sum += data[n - 1 - i]; }

    // LOOP 3: compute sum of squares
    double sum_sq = 0.0;
    for (size_t i = 0; i < n; ++i) { sum_sq += data[i] * data[i]; }

    double end = omp_get_wtime();
    double mean = sum / (double)n;
    double mean_sq = sum_sq / (double)n;
    double variance = mean_sq - mean * mean;

    printf("variance       = %.2f\n", variance);
    printf("total time (s) = %.2f\n", end - start);
    printf("num threads    = %d\n\n", omp_get_max_threads());

    free(data);
    return 0;
}
```