Exercise 5: OpenMP and miniHPC

Wenjing Wang

Task 1: Connecting to miniHPC and Using Slurm

1. Connecting to miniHPC

I connected to the miniHPC login node using:

```
ssh wang0041@cl-login.dmi.unibas.ch
```

2. Submitting a simple Slurm job

I created the following job script:

```
I submitted jobscript using:
\begin{verbatim}
sbatch jobscript
Submitted batch job 2454288
```

3. Output of the job

The Slurm output file `slurm-2454288.out` contained:

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

Therefore, the allocated compute node was: **dmi-cl-node003**.

Task 2: Investigating Memory Architecture

1. Investigating the Xeon node memory architecture

To investigate the architecture, I requested an interactive session on a Xeon compute node:

```
srun --partition=xeon --ntasks=1 --cpus-per-task=1 --pty bash
```

This placed me on:

```
dmi-cl-node003
```

Then I ran:

```
lscpu
```

Important results from `lscpu`:

- 20 CPUs in total
- 10 cores per socket
- 2 sockets
- 2 NUMA nodes

- NUMA node 0: CPUs 0–9
- NUMA node 1: CPUs 10–19
- CPU model: Intel Xeon E5-2640 v4 @ 2.40GHz

Next, I loaded hwloc and ran `lstopo`:

```
module load hwloc
lstopo --no-io --of txt
```

From `lstopo`, I observed:

- Total memory: 63 GB
- NUMA Node 0: 31 GB
- NUMA Node 1: 31 GB
- Each NUMA node corresponds to one physical CPU socket

2. Memory architecture in lecture terms

In terms of memory architecture discussed during the lecture:

- Each miniHPC Xeon node is a **shared-memory multiprocessor system with a NUMA (Non-Uniform Memory Access) architecture**.
- More specifically, it is a **ccNUMA system**: all cores share a global address space, but local memory access is faster than remote access.
- The entire miniHPC cluster as a whole is a **distributed-memory system** consisting of multiple NUMA nodes connected via a network.

3. NUMA domain definition and number of domains

A **NUMA domain** (or NUMA node) is a group of CPU cores that share a local memory region with low and uniform access latency. Access to memory inside the same NUMA node is fast, while access to memory in other NUMA nodes is slower.

From the results above:

- One Xeon compute node contains **2 NUMA domains**.
- NUMA Domain 0: CPUs 0–9 with 31 GB memory.
- NUMA Domain 1: CPUs 10–19 with 31 GB memory.

Thus, a single miniHPC Xeon node has **two NUMA domains**.

Task 3: "Hello World" OpenMP and Job Configuration

C Program

The following OpenMP program prints information about hardware resources and thread identifiers:

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

int main(void) {
    int num_procs   = omp_get_num_procs();
    int max_threads = omp_get_max_threads();
```

```
    printf("omp_get_num_procs()   = %d\n", num_procs);
    printf("omp_get_max_threads() = %d\n", max_threads);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        printf("Hello from thread %d of %d\n", tid, num_threads);
    }
    return 0;
}
```

Slurm Job Script

```
#!/bin/bash
#SBATCH --job-name=hello_omp
#SBATCH --output=hello_%j.out
#SBATCH --partition=xeon
#SBATCH --nodes=1

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

echo "Job ID: ${SLURM_JOB_ID}"
echo "Node list: ${SLURM_NODELIST}"
echo "ntasks-per-node: ${SLURM_NTASKS_PER_NODE}"
echo "cpus-per-task: ${SLURM_CPUS_PER_TASK}"
echo "OMP_NUM_THREADS: ${OMP_NUM_THREADS}"

./hello_openmp
```

Job Configurations and Observations

I submitted four configurations using `sbatch`:

Case	ntasks-per-node	cpus-per-task	Processes	Threads/proc	Expected Hello lines
1	1	1	1	1	1
2	1	2	1	2	2
3	2	1	2	1	2
4	2	2	2	2	4

The observed outputs match the expected behavior, noting that Slurm may merge stdout from multiple tasks, so only a subset of the lines may physically appear in the output file.

Explanation of Output Behavior

The total number of "Hello" lines equals

$$\text{processes} \times \text{threads per process}.$$

Each Slurm task corresponds to one process, and each process launches `OMP_NUM_THREADS` OpenMP threads. Since multiple threads and tasks print concurrently, their output is interleaved and not ordered. Slurm does not guarantee deterministic stdout ordering across tasks.

Task 4: Data Parallelism through Work Sharing Loops

The program from Listing 2 was parallelized using four different approaches, all executed on miniHPC with **8 OpenMP threads**. Execution times were measured using `omp_get_wtime()`.

1. Execution time of the sequential version

```
Sequential time: 0.550 sec
```

2. Manual SPMD parallelization

A single `#pragma omp parallel` region was used, and each thread manually computed its chunk, identified via `omp_get_thread_num()`. Partial results were summed afterwards using atomics.

```
Manual SPMD time: 0.065 sec
```

3. Work-sharing using `omp for` and reduction

Each loop in Listing 2 was parallelized with a separate `#pragma omp parallel for` and a `reduction` clause.

```
parallel for (reduction) time: 0.066 sec
```

4. Work-sharing with `nowait`

The `nowait` clause removes the implicit barrier at the end of each `omp for` loop. Since the loops are placed in different parallel regions, eliminating the barrier does not significantly change performance.

```
parallel for (reduction, nowait) time: 0.065 sec
```

5. Comparison and discussion

Implementation	Time (sec)
Sequential	0.550
Manual SPMD	0.065
`omp for` + reduction	0.066
`omp for` + reduction + nowait	0.065

Discussion. The sequential version is by far the slowest, since only one CPU core is utilized. All three parallel versions achieve an almost ideal speedup of approximately 8×.

The manual SPMD version is slightly faster than the others because the program enters an `omp parallel` region only once, reducing thread-creation overhead and enabling good cache locality. The `omp parallel for` version is nearly as fast, with minimal overhead introduced by creating multiple parallel regions.

Adding the `nowait` clause does not notably change the execution time. Since the loops belong to different parallel regions, the implicit barriers cannot be avoided entirely, so the potential benefits of `nowait` do not apply in this case.

Task 5: Scaling OpenMP and Data Access Patterns

We executed the parallelized OpenMP program using different thread counts (cpuspertask $\in \{1, 2, 4, 8, 16\}$) on a miniHPC Xeon compute node. The "Manual SPMD" version was chosen for measurement, although all parallel variants showed similar trends.

Execution Times

Threads	Time (sec)
1	0.384
2	0.194
4	0.099
8	0.066
16	0.110

Discussion

The execution time decreases substantially when increasing the number of threads from 1 to 8. This region shows almost linear speedup, since the application is embarrassingly parallel and the compute node has sufficient CPU resources to keep all threads busy.

However, using 16 threads results in *worse* performance. This behaviour is expected for memory-bound workloads. All loops in the program primarily perform simple arithmetic on a large array (approximately 800 MB), and hence execution is limited by memory bandwidth rather than by compute capacity. Once the available memory bandwidth is saturated (around 8 threads), additional threads do not increase throughput.

Furthermore, the Xeon node contains two NUMA domains. Running with 16 threads forces threads to access memory allocated on both sockets, introducing higher latency and non-local memory traffic. Cache contention and OpenMP scheduling overheads additionally contribute to the slowdown.

In summary, the execution time does *not* scale proportionally to the number of threads. Scaling is strong up to the point where memory bandwidth becomes the dominant bottleneck (around 8 threads), after which performance degrades.