Exercise Sheet 4 for Lecture Parallel Programming

Deadline: 2026-05-24, 23:59

Prof. Dr. Michael Kuhn (michael.kuhn@ovgu.de)
Michael Blesel (michael.blesel@ovgu.de)
Parallel Computing and I/O • Institute for Intelligent Cooperating Systems
Faculty of Computer Science • Otto von Guericke University Magdeburg
https://parcio.ovgu.de

We start with our serial program for solving the Poisson equation. However, we are only looking at the Jacobi method for now. We will create parallel versions of this program with the help of OpenMP. In the materials you can find an optimized serial version of `partdiff` which you shall use for the following parallelization tasks.

With the `-fopenmp` option, GCC will create OpenMP code. An additional tutorial for programming with OpenMP can be found at:

https://hpc-tutorials.llnl.gov/openmp/

1. Parallelization with OpenMP (120 Points)

Parallelize the Jacobi method from the serial program with OpenMP. The parallel version must deliver the same results as the serial one. The termination after precision as well as the termination after iteration count must work correctly and deliver the same output as the serial version! Use the OpenMP clause `default(none)` to prevent errors.

The reachable speedup is dependent on the concrete hardware being used, but with 1,024 interlines you should observe a noticeable speedup.

The thread count shall be controllable via the already existing first argument of the `partdiff` application.

When you run your program you can for example use the `top` tool (press "1" to see all the individual cores) to watch the processor usage.

Please take notes on how much time you needed for this task. How much of that time was spent looking for errors?

2. Data Distribution (120 Bonus Points)

In addition to the default data distribution, distribute the data in three different ways:

- Row-wise distribution (thread 1 is assigned the first line(s), …)
- Column-wise distribution (thread 1 is assigned the first column(s), …)
- Element-wise distribution (every matrix element can be calculated by a different thread)

Implement the different data distributions in your program. Each version shall have their own version of the `calculate` function. If multiple data distributions are implemented, submit the code in a way that each data distribution creates its own binary via the Makefile. For this, you can use `-D` to set flags during compilation. The macros defined with `-D` can be used in the code in the following way:

```
#ifdef ELEMENT
...
#endif
```

In the materials, you will find an already-modified `Makefile` and a `partdiff.c` with four `ifdef` clauses (`DEFAULT` for task 1 and three additional clauses for task 2).

Hint: For this task, you might need to modify the code inside the loops.

Perform measurement 1 from task 3 for all of the data distributions and write down explanations for the obeserved differences.

3. Performance Analysis (180 Points)

For this task, the performance of the parallelized version shall be analyzed. The parallelized program should reach a speedup of at least 10 when run with 24 threads and 4,096 interlines.

Repeat each measurement at least three times so that you can calculate significant averages. For this, you can use the tool `hyperfine` which you can load with `module load hyperfine`.

For `t` threads, `i` interlines and `n` iterations you can use the following command:

```
./partdiff t 2 i 2 n
```

The measurements should be done with SLURM on one of the compute nodes of the cluster. Document the hardware specifications used for the measurements (processor, core count, available main memory, etc.) and use the same compute node for each batch of measurements.

In the materials, you can find prepared SLURM job scripts within the `slurm` directory, which you can use for your measurements. To run them, your working directory should be the `slurm` directory from which you can start the scripts with `sbatch`:

```
$ cd PP-2026-Exercise-04-Materials
$ make -C pde
$ cd slurm
$ sbatch measurement1.slurm
$ sbatch measurement2.slurm
```

Measurement 1

Figure out the performance of your OpenMP programs and compare the runtimes for 1–24 threads in a diagram. Importantly, also compare your version with the original serial program! Use 4,096 interlines for your measurements. The shortest run should take at least 30 seconds; choose fitting parameters.

Run these measurements for the default distribution as well as the additional data distributions from task 2. What reasons might there be for the results of the measurements? Write down you explanations (roughly 1 page).

Measurement 2

Now determine how your OpenMP program scales depending on the matrix size (interlines). For this, use 24 threads and 13 measurements between 1 and 4,096 interlines (where interlines = 2^i for $0 \leq i \leq 12$). The longest run should take about 30 minutes. Start with 4,096 interlines and choose fitting parameters; for the following runs you can then reduce the amount of interlines via the given formula. Visualize all results in fittingly labeled diagrams.

Only run this measurement for the default data distribution. Write roughly half a page of interpretation of the results.

4. Parallelization Scheme (180 Points)

The goal of the coming exercise sheets will be to parallelize the `partdiff` application via POSIX Threads and MPI. But first, you will create a parallelization scheme.

You will need to distribute the matrix data over threads or processes (called "tasks" in the following). That means that each tasks will work on a part of the data. Keep the following in mind: To calculate the values of one row, you will always need the values from the row above and the row below (see the following figure).

···	`Matrix[i - 1][j]`	···
`Matrix[i][j - 1]`	`Matrix[i][j]`	`Matrix[i][j + 1]`
···	`Matrix[i + 1][j]`	···

Therefore, if the currently calculated row is the first or last one of the data block of a task, the needed neighboring row will be managed by a neighboring task. The neighbor task might have to share the data. Also, after calculating a border row, a task might have to share the new results with its neighbor as well. The problem is that this has to happen at the correct time with the correct values. Depending on whether shared or distributed memory is used, the way of exchanging border lines will differ. Also think about which tasks need to access which variables and if conflicts could arise because of that.

Also think about the differences of the calculation with the two different methods (Jacobi Gauß-Seidel). Each method might require different communication schemes. Furthermore, think about the first and last tasks which only have one neighboring task each.

To optimize the memory usage, in the case of distributed memory, each task may only hold the data of their own sub-matrix in memory. This allows for problems to be computed that would not fit inside the main memory of a single node.

Also discuss problems that may occur regarding the termination conditions of the program. For each of the cases, give a brief description about what kinds of problems could arise and how you would solve them (in case that the problem seems solvable).

Discuss at least the following points in detail:

1. Description of the data distribution of the matrix for the multiple tasks
    - Which part of the matrix gets managed by which task?
    - Visualize the data distribution with a suitable diagram.
2. Parallelization scheme for the Jacobi method
    - From the perspective of a task, describe at what time the calculation and the communication with its neighbors take place. Differentiate between shared and distributed memory.
    - What data does the task require from its neighbors and when will the data exchange take place?
    - Which variables and data do need to be accessed by which task?
3. Parallelization scheme for the Gauß-Seidel method (see Jacobi)
4. Discussion of the termination problem
    - There are four cases to look at: termination after number of iterations and precision, for Jacobi and Gauß-Seidel each.
    - At which time will a task notice that the termination condition has been reached and it can stop its calculations?
    - In which iteration will the other tasks be at the time that a task notices that the termination condition has been reached?

Submission

We will count your last commit on the `main` branch of your repository before the exercise deadline as your submission. In the root directory of the repository, we expect a `PP-2026-Exercise-04-Materials` directory with the following contents:

- A file `group.md` with your group members (one per line) in the following format:

    ```
    Erika Musterfrau <erika.musterfrau@example.com>

    Max Mustermann <max.mustermann@example.com>
    ```

- The modified code of the `partdiff` application in the `pde` directory (Tasks 1–2)

  - A `Makefile` with the targets `partdiff-{element, row, column}` for binaries `partdiff-{element, row, column}`, which implement the corresponding data distributions (Task 2)

- An elaboration `performance-analysis.pdf` with the measured runtimes and performance analysis (Task 3)
- A file `parallelization-scheme.pdf` with your answers (Task 4)