Exercise Sheet 3 for Lecture Parallel Programming

Deadline: 2026-05-10, 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

---

1. SLURM Usage (20 Bonus Points)

After logging in via SSH, you are on the so-called login node of the cluster. This is primarily intended for managing your data and compiling software. More complex tasks should always be performed on compute nodes.

The cluster uses SLURM for job management, which can be used to gain access to the compute nodes. A distinction must be made between non-interactive and interactive use. In the non-interactive case, you submit so-called jobs, which SLURM executes on the compute nodes. In the interactive case, you reserve a compute node and log in to it.

You can display all available partitions and nodes with `sinfo`. The partition used for the lecture and exercises is called `vl-parcio`. For interactive use, nodes can be reserved with `salloc`, which starts a subshell that keeps the reservation until you leave the subshell with `exit`.

```
$ salloc -p vl-parcio
$ srun --pty bash
```

Both commands can also be combined for a temporary allocation.

```
$ srun -p vl-parcio --pty bash
```

By default, a single core (with two threads) is allocated. Additional cores can be requested with the `-c` parameter, specifying the number of threads. For example, a value of 4 corresponds to two cores with two threads each. The allocated core or thread count can be easily checked with the `nproc` command.

```
$ srun -p vl-parcio -c 4 nproc
```

Question: Why are four threads allocated if you request three with the `-c` parameter? Write a few sentences to explain this behavior.

A non-interactive job can be submitted using `sbatch job.slurm`. Active jobs can be displayed with `squeue`; output can also be restricted to own jobs with `squeue --me`. If necessary, a job can also be canceled with `scancel <job-id>`. The following script can serve as a template.

```
1 #!/bin/bash
2
3 # Time limit is ten minutes (see "man sbatch")
4 #SBATCH --time=10:00
5 # Run one task on one node
```

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
# Make 48 cores available to our task
#SBATCH --cpus-per-task=48
# Use vl-parcio partition
#SBATCH --partition=vl-parcio
# Redirect output and error output
#SBATCH --output=job.out
#SBATCH --error=job.err

srun hostname
srun nproc
srun ./my-application
```

By default, one task is allocated on one node. The number of tasks can be adjusted with the parameter `-n` (or `--ntasks`) and the number of nodes can be adjusted with the parameter `-N` (or `--nodes`). Both parameters become necessary only when programming with MPI.

2. Parallel Execution of a Shell Script (40 Points)

1. Create a shell script `timescript.sh` which creates the following output:

   `HOSTNAME: TIMESTAMP`

   `HOSTNAME:` Short hostname of the node used to execute the script.
   (Hint: `hostname --short`)

   `TIMESTAMP:` Timestamp of the time when the script is executed with a precision of at least micro seconds.
   (Hint: `date --iso-8601=ns`)

   (Hint: Take a look at the man pages of `hostname` and `date`.)

2. Create a job script `jobscript.sh`, which starts `timescript.sh` in parallel on four nodes with four processes each. The output shall be in a file `timescript.out` that contains the output of *every* execution of `timescript.sh`. The script shall be usable with `sbatch jobscript.sh`.
   (Hint: Take a look at `srun`, `#SBATCH -N` and `#SBATCH -n`.)

3. After `jobscript.sh` has written the `timescript.out` file it shall output "done". Modify the script in a way so that "done" is written to a file called `jobscript.out`.

4. Run the script multiple times.

   - **Question:** What do you notice from the timestamps? Try to explain your observations!
   - **Question:** Could the `timescript.out` file also be created inside the `timescript.sh` script? If so: How? If not: Why not?

3. Performance Optimization (250 Points)

We want to solve the Poisson equation via an iterative method:

We imagine ourselves in a typical situation of many PhD students that enter the domain of parallel programming: You are given the code of a serial program. The code has a number of lines in the seven digits, it is created in a programming language unknown to you, additionally it is uncommented and the original programmer is no longer with us. But at least you have the sources (see materials for this sheet).

The sources include the complete source code, a bit of additional information and a program called `partdiff`. Use the program to test the included Poisson solver. Play around a bit with the parameters and take a look at the included reference solutions.

The problem size N is controlled via the `interlines` variable and calculated with the following formula: $N = 9 + (8 \times \texttt{interlines}) - 1$. This allows the `DisplayMatrix` function to deliver a comprehensible output. It always returns exactly 9 rows and columns. Use this output to verify the correctness of your modfications.

Included in the serial code are two algorithms to solve the Poisson problem (see Section 4.4): The Jacobi and Gauß-Seidel methods. An overview about the mathematical background is given in the appendix of this exercise sheet. It is not required to read or understand the mathematical background to solve the tasks on this exercise sheet, it is there in case you are interested.

In the code there are some performance problems that you are supposed to find and fix. The program should be faster by a factor of approximately 10 after your modifications. Work on the following steps to increase the performance of the program. Measure and document the performance increase after each step.

1. Profile the program with the help of `gprof` to find time intensive functions that are candidates for performance optimizations.
2. Analyze and optimize the memory access patterns.
3. Compile the program with different compiler optimzation flags.

Your modifications should mainly be inside the `calculate` function. You do not need to include optimizations which are automatically done by the compiler (for example, elimination of unneccessary variables, reordering of lines of code etc.).

To use `gprof` modify the `Makefile` to compile `partdiff` with the `-pg` option. If you now run the program, a `gmon.out` file will be created. Use `gprof ./partdiff` to view the recorded data. Explain the output.

Document your changes to the code and measure for each change the difference in runtime. You can get the runtime of the program with the `time` command (`time ./partdiff ...`). Run `partdiff` with the following parameters:

```
./partdiff 1 2 64 2 5120
```

Important: The output has to be the same before and after your modifications! In the `reference` directory of the partdiff program, you can find reference results for these parameters.

3.1. Mathematical Optimizations (30 Bonus Points)

Think about how the mathematical operations could be done in a more optimal way. This does not require you to change the algorithms!
(Hint: Take a look at the mathematical calculations and think about whether they could be achieved in a faster way)

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-03-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>
  ```

- A file `slurm-answers.md` with your answers (Task 1)
- A file `timescript-answers.md` with your answers, the following scripts that can be run on the cluster `timescript.sh` and `jobscript.sh` as well as a file `timescript.out` with the output of one run of your script with multiple processes (Task 2)
- A file `pde-optimizations.md` with your results (Task 3)
  - Describe your optimizations and the achieved performance increase
  - Output of `gprof` with explanations
  - The modified code of the `partdiff` program in the `pde` directory

4. Mathematical Background

This section is supposed to help interested students to understand the background of the program. In case you have problems following the algorithms used in the code, the following text could help you gain a deeper understanding. **However, it is not necessary to understand the mathematical background to solve the tasks on this sheet.**

Many natural and technical processes can be described with partial differential equations. An example for this is the Poisson equation. Due to the lack of analytical solutions it is often necessary to use methods from numerical mathematics.

You get to a system of linear equations via the use of:

- Discretization (determination of interesting points in the solution space)
- Exchange of the differential quotients with difference quotient

To caluclate the solution vector of this system there are direct and indirect methods. Due to the drawbacks of direct methods (elimination methods such as Gauß elimination), namely a too high algorithmic complexity and also numerical instability, indirect methods are currently preferred. We are looking at two iterative methods for the `partdiff` program.

4.1. Problem Statement

Given is a partial differential equation of the form:

$$-u_{xx}(x, y) - u_{yy}(x, y) = f(x, y) \text{ mit } 0 < x, y < 1 \tag{1}$$

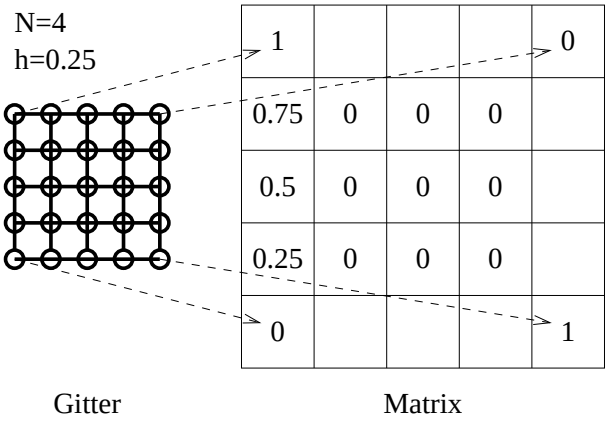
This representaton is called the Poisson problem. u_{ii} is the second derivative of the function u by i. The function $f(x, y)$ is called the error function. The boundary values $u(\_, 0)$, $u(\_, 1)$, $u(0, \_)$ and $u(1, \_)$ are known. We are looking for $u(x, y)$ for $0 < x, y < 1$.

4.2. Discretization

A solution for the Poisson equation shall be calculated on the area $[0, 1] \times [0, 1]$. The easiest discretization is a equidistant quadratic grid.

Number of intervals in every direction: N
Number of points in the whole area (with borders): $(N + 1)^2$
Number of inner grid points: $(N - 1)^2$
Grid width h: $1/N$

This grid can be stored in a $(N + 1) \times (N + 1)$ matrix. Every entry in this matrix represents a point in the grid. The boundary values of the grid are set before the start of the calculations.



4.3. Transition From Differential to Difference Quotient

We define the inner grid points.

$$u_{i,j} := u(i * h, j * h) \text{ mit } i, j = 1, ..., N - 1 \tag{2}$$

The substitution of the partial derivative from (1) with the finite differences of second order:

$$u_{xx;i,j} := 1/h^2(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})$$

$$u_{yy;i,j} := 1/h^2(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})$$

This yields a system of linear equations of the form:

$$1/h^2(4v(i,j) - v(i-1,j) - v(i+1,j) - v(i,j-1) - v(i,j+1)) = f(i,j) \tag{3}$$

Method of Solving the System of Linear Equations

Representation of (3) in matrix form: $Au = h^2 f$ (Derivation and structure of the matrix A is irrelevant for the application of the iterative method.)

Exact solution: u
Approximation: v (Shall be calculated iteratively until v only differs minimally from u.)
Error: $e = u - v$
Sadly, u ist unknown, therefore e is not determinable.
But, determinable are:

- Residuum $r := h^2 f - Av$ (r is the magnitude of which the approximation of v is apart from the original problem $Au = h^2 f$.)
- Norm of the residuums $\| r \|_\infty := \max | r(i,j) |$
- Correlation: $\| r \|_\infty = 0 \iff e = 0$

From $Au = h^2 f$ and $Av = h^2 f - r$ we get the subtraction $Ae = r$, or $e = A^{-1} r$, called the residuum equation.

To calculate an approximation $\hat{e}$ for e: In the residuum equation, instead of A, use the easily invertable matrix $D = (d_{i,j})$ with $d_{i,j} = 4\delta(i,j)$ (δ: Kronecker symbol).

D is the matrix that results from A by zeroing all non principal diagonal elements.

Approach:

1. Calculate or guess initial v^0 (set to 0 for the sake of convenience). Set $i = 0$.
2. Set $i = i + 1$. Calculate r^i via $r^i = h^2 f - Av^{i-1}$.
3. Calculate $\hat{e}$ via $\hat{e} = D^{-1}r$.
4. Calculate new approximation $v^i = v^{i-1} + \hat{e}$.
5. If $\| r \|_\infty <$ bound, abort, else to 2.

4.4. Iterative Solution Methods

Principal: Guess first approximation, then improve it iteratively.

- Jacobi method: Use two matrices for v: Updating the new values is done by inspecting the old values.
- Gauß-Seidel method: Use only one matrix for v, which means that new values are used as soon as they have been calculated.

4.5. Pragmatic Approach

Scheme for the program to solve the Poisson equation:

```
Initialize matrix (boundaries and innner points)
while (maximum_residuum > bound)
    for all lines DO
      for all columns DO
         // Calculate the stencil
         star =                    - v[m_old][x][y+1]
               - v[m_old][x-1][y] + 4*v[m_old][x][y] - v[m_old][x+1][y]
                                   - v[m_old][x][y-1];
         // caluclate the compenstation value
         correction = (f(h*x,h*y) * h_square - star) / 4;
         // Form termination condition
         Calculate norm of residuum
         Calculate maximum_residuum
         // New matrix values
         v[m_new][x][y] = v[m_old][x][y] + correction;
         // Gauß-Seidel: m_new = m_old
```

Literature

- William H. Press: Numerical Recipes in Pascal/C/Fortran, Cambridge, USA 1990