Examination 1—OpenMP

Regulations

Groups

The tasks must be completed together with your team partner. However, the submission must clearly indicate which student authored which parts of the solution. It is sufficient to include a short comment inside the source code, and marking sections as "created together" is also appropriate. Moreover, each team partner must be able to understand and explain **all** parts of the submitted solution.

Solutions and ideas must **not** be exchanged with other groups. If identical or nearly identical solutions are submitted by different groups, all affected submissions maybe marked as failed. Submissions will also be compared against solutions from previous semesters.

Questions

If you have content-related or organizational questions, please post them in the discussion forum on the e-learning platform. Make sure not to reveal any details of your own solution. If posting publicly is not possible in exceptional cases, please send an email to jonas.posner@cs.hs-fulda.de.

Cluster

All programs must run correctly on the HPC cluster of Fulda University of Applied Sciences. For all measurements, the compiler `gcc/14.3.0` must be used.

Reproducibility

For each program, you must provide the complete command lines for compilation and execution (including any required environment variables), as well as all Slurm scripts used to run your jobs. Please name all files exactly as specified in the assignment. In addition, submit all Slurm output files containing the measured times used in your speedup tables. Non-reproducible solutions may be marked as failed.

Submission format

Please submit a single `.zip` file via the e-learning platform. The archive must contain:

- all `.c` files (your implementations),
- the `Makefile`,
- all Slurm scripts used for running the programs,
- all Slurm output files containing the measured execution times,
- a `README.md` with instructions for compilation and execution,
- a PDF document containing your speedup plots and tables.

Submissions that do not follow this structure may be marked as failed.

Submission deadline

The solutions must be submitted no later than 23rd January 2026, 09:00 AM via the e-learning platform. Submissions received after this deadline will be treated as not submitted.

Usage of third-party sources

All external sources used (e.g., websites, books, articles, presentation slides, etc.) must be cited. Solutions created fully or partially with the help of AI tools (e.g., ChatGPT) must also be referenced accordingly. Failure to properly cite sources will be considered plagiarism and may be regarded as an attempt to deceive by the entire group. This applies in particular to code.

Please submit your solutions by **23rd January 2026, 09:00 AM** !

Task 1.1 (`heatmap_analysis.c`)

Write an *efficient* OpenMP program that analyzes a two-dimensional array that is interpreted as a so-called *heatmap*. Take into account all efficiency aspects discussed in the lecture.

Create an `unsigned long` array of size `rows` · `columns`. To ensure comparability of the results, the array must be initialized as follows (your C code may differ as long as the resulting values are identical):

```
for (int i = 0; i < rows; ++i) {
  for (int j = 0; j < columns; ++j) {
    unsigned long s = seed * concatenate(i, j);
    A[i][j] = my_rand(&s, lower, upper);
  }
}
```

Implementations of both the `my_rand` and `concatenate` functions are given on the last page of this sheet. `rows`, `columns`, `seed`, `lower`, and `upper` are program parameters (see below).

The analysis consists of two sub-tasks:

Part 1: Range sums For each column x, the maximum sum of a vertical window of height `window_height` (h) is to be determined. For this, the sum over the interval $[i, i+h-1]$ for $i = 0$ to $rows - h$ must be computed.

Part 2: Local hotspots A value $A[i][j]$ is called a *local hotspot* if it is strictly greater than all of its direct neighbors (up, down, left, right). Output, for each row, the number of local hotspots.

Attention: Every value used in the analysis must be transformed `work_factor` times before it is used in any computation. The transformation is performed via the `hash` function, which is given on the last page of this sheet. Example:

```
for (int w = 0; w < work_factor; ++w)
  A[i][j] = hash(A[i][j]);
```

At the end, the program must output the following information:

- The maximum sliding sums per column (if `verbose=1`),
- the number of hotspots per row (if `verbose=1`) and the total number of hotspots (always),
- the total execution time (always).

The output must be formatted **exactly** as in the corresponding examples. Whether the output itself is parallelized is up to you. The computations must always be performed—independent of the value of `verbose`.

Program invocation:

```
./heatmap_analysis columns rows seed lower upper window_height verbose
                   num_threads work_factor
```

`columns`	Number of columns in the array
`rows`	Number of rows in the array
`seed`	Initial value for the random number generator
`lower`, `upper`	Value range for random numbers (lower bound inclusive, upper bound exclusive)
`window_height`	Window height for the vertical range analysis
`verbose`	if set to 1, the array, the maximum range sums per column, and the number of hotspots per row are printed. If set to 0, only the total number of hotspots is printed
`num_threads`	Number of OpenMP threads
`work_factor`	Number of transformations per value

Example 1:

```
./heatmap_analysis 3 4 42 0 10 2 1 1 1

Starting heatmap_analysis
Parameters: columns=3, rows=4, seed=42, lower=0, upper=10, window_height=2,
verbose=1, num_threads=1, work_factor=1

A:
0, 0, 7
9, 9, 8
2, 4, 5
7, 3, 4

Max sliding sums per column:
8078608628849563118, 17805128741409388388, 17804482663663377439

Hotspots per row:
Row 0: 1 hotspot(s)
Row 1: 0 hotspot(s)
Row 2: 2 hotspot(s)
Row 3: 0 hotspot(s)

Total hotspots found: 3
Execution took 0.0020 s
```

Example 2:

```
./heatmap_analysis 1024 786 1337 0 100 20 0 1 10

Starting heatmap_analysis
Parameters: columns=1024, rows=786, seed=1337, lower=0, upper=100, window_height=20,
verbose=0, num_threads=1, work_factor=10

Total hotspots found: 159599
Execution took 0.0958 s
```

Example 3:

```
./heatmap_analysis 1024 786 123 0 100 55 0 1 100

Starting heatmap_analysis
Parameters: columns=1024, rows=786, seed=123, lower=0, upper=100, window_height=55,
verbose=0, num_threads=1, work_factor=100

Total hotspots found: 151963
Execution took 0.5967 s
```

Note: The outputs were generated on the Fulda HPC Cluster. If you compile or run your program with other compilers or platforms (e.g. Windows), values printed for `unsigned long` may differ. To ensure consistent results across platforms, consider using `uint64_t` instead. However, it is crucial that it runs correctly on the Fulda HPC Cluster!

Speedup measurement: Determine speedups for your program for 2, 4, 8, 16, 32, and 64 threads. Perform the measurements on the Fulda HPC Cluster and submit the measured values and speedup values (including a graphical plot). `verbose` must be set to 0; the remaining parameters should be chosen appropriately. The time measurement must start immediately after reading the command-line parameters and end immediately after the output.

Task 1.2 (`heatmap_analysis_quick.c`)

Write a second OpenMP program that performs the same analysis as in Task 1.1, but with a different objective: As soon as at least one row without a local hotspot is found, the program should terminate as quickly as possible.

As soon as such a case is detected, the program should output an appropriate message and terminate immediately, without checking any further rows.

In the success case (i.e., when *all* rows contain at least one hotspot), the program should, as in Task 1.1, perform the full analysis.

Example 4:

```
./heatmap_analysis_quick 3 4 42 0 10 2 1 1 0

Starting heatmap_analysis
Parameters: columns=3, rows=4, seed=42, lower=0, upper=10, window_height=2,
            verbose=1, num_threads=1, work_factor=0

Row 2 contains no hotspots.
Early exit.
Execution took 0.0002 s
```

Note: which row without a local hotspot is found first may vary.

Speedup measurement: Determine speedups for this program as well for 2, 4, 8, 16, 32, and 64 threads and submit the measured values and speedup values (including a graphical plot).

Task 1.3 (`pi_tasks.c`)

Write an efficient OpenMP program that computes the value of π multiple times using the integral method discussed in the excercises.

For the parallelization, *OpenMP tasks* must be used:

- Start with a single task that computes π.
- After finishing its computation, each task should *randomly* create between 1 and 4 new tasks, as long as the total number `num_tasks` is not exceeded.
- Each task should compute π with its own precision, which is *randomly* chosen in the range `[lower, upper]`.
- All random numbers must be generated *deterministically* using a seed.

At the end the program should print the following information:

- The average of all computed π values,
- for each thread the number of tasks processed by that thread,
- the total execution time of the computation.

Example invocation:

```
./pi_tasks num_tasks num_threads lower upper seed
```

Example:

```
./pi_tasks 1000 4 100000 1000000 42

Average pi: 3.1415926536
Thread 0 computed 239 tasks
Thread 1 computed 249 tasks
Thread 2 computed 258 tasks
Thread 3 computed 254 tasks
Execution took 2.3739 s
```

Speedup measurement: Run your program with the following configuration:

```
./pi_tasks 20000 1 10000 1000000 42
```

Determine speedups for your program using 2, 4, 8, 16, 32, and 64 threads. Perform the measurements on the Fulda HPC cluster and provide the measured times and speedup values (including a graphical plot). The timing should start immediately after reading the command-line parameters and end immediately after printing the output.

my_rand:

```
unsigned long my_rand(unsigned long* state, unsigned long lower, unsigned long upper) {
    *state ^= *state >> 12;
    *state ^= *state << 25;
    *state ^= *state >> 27;
    unsigned long result = (*state * 0x2545F4914F6CDD1DULL);
    unsigned long range = (upper > lower) ? (upper - lower) : 0UL;
    return (range > 0) ? (result % range + lower) : lower;
}
```

concatenate:

```
unsigned concatenate(unsigned x, unsigned y) {
  unsigned pow = 10;
  while (y >= pow)
  pow *= 10;
  return x * pow + y;
}
```

hash:

```
unsigned long hash(unsigned long x) {
  x ^= (x >> 21);
  x *= 2654435761UL;
  x ^= (x >> 13);
  x *= 2654435761UL;
  x ^= (x >> 17);
  return x;
}
```