Assignment1: Heatdiffusion openMP and MPI

Given is a C program for implementing the approximation of a simple heat diffusion scheme in 2D (five-point stencil).

It iteratively computes matrices of size $n * n$ from an initial $n * n$-matrix That matrix has all values set to `0.0` but the two values in the first row at indeces $[0, n/2]$ and $[0, (n * 3)/4]$. These values are initialised by heat values `100.0` and `1000.0`, respectively. The iteration step recomputes all non-boundary elements by computing a weighted sum of the current values of the element itself and its four immediate neighbouring elements. The number of iterations is provided as a parameter of the program.

This assignment explores how this algorithm can be executed on a multicore system using two different technologies: openMP and MPI.

Your tasks are to:

- explore how to optimise the sequential program
- develop an openMP version and explore the effectiveness of your version
- develop an MPI version and explore the effectiveness of your version
- provide an extensive performance analysis

This assignment should be done by teams of 4 students. How you distribute the work within the team is up to you. However, you need to declare who did which part. Ideally, you perform all performance measurements on a single system. If that constitutes a logistic problem, you can use different hardware provided you clearly specify which hardware has been used for which measurements.

Make sure that you:

- specify exactly what hardware is being used (CPU version, clock frequency, memory, etc.)
- specify exactly what software is being used (compiler version, compiler flags, etc.)
- specify exactly which parameter (size n and number of iterations) you are using
- repeat each experiment at least 5 times and report average time as well as the variability (error bars)

Task 1: Sequential Evaluation

Evaluate the performance of the sequential code. Use the highest level of compiler optimisation on your machine. Typically, this is `-O3` but you should look into the man pages of your compiler; Look for other flags such as `-march=native`. Present the wallclock time of the work-loop as a function of the matrix-size $n * n$. Present the absolute performance in `GFLOP/s` as a function of the matrix-size $n * n$. Make sure that you vary the parameter n of your matrices so that the sizes of your matrices vary from a few kB to something as large as 5GB. Note that one `double` requires 8 bytes. You may have to adjust the iter parameter for your machine to obtain a reasonable range of runtimes.

Analyse any anomalies this function may show. You may want to use tools such as gprof, valgrind, or the gperftools to find out what is going on. Summarize your findings in a few paragraphs.

Try to improve your program. Ideas: you may want to find out whether the compiler you use manages to vectorise the code. `gcc` offers compile time information about this (`-fopt-info-vec` or similar). Alternatively, you may want to look at the generated assembly (`gcc -S`). If the compiled code is not vectorised, you may want to try out whether you can rewrite the inner loops in some way to help the compiler succeed. Analyse the impact of your optimisations by re-running your experiments.

Task 2: openMP

Add openMP pragmas and library calls to your sequential C versions (non-optimised or optimised). Repeat the evaluations from Task 1 for the openMP version with different numbers of threads. Make sure that you instruct slurm to provide you with sufficient cores and sufficient memory!. Run several experiments to figure out what impact the different scheduling strategies have and present all these in a single graph. Again, try to analyse the performance and try to explain your findings.

Task 3: MPI

Rewrite the program to leverage MPI. Try to optimise your MPI version considering both sequential versions. Repeat the evaluations from Task 1 for your MPI versions quantifying the effects that your own optimisation attempts have (scheduling, placement, communication pattern, etc.). Make sure that you request sufficient memory for the number of MPI ranks that you attempt to allocate.

Task 4: Performance

Provide a discussion of your overall findings. This should include figures reporting speedups, scaling (strong and weak), and efficiency. Furthermore, you should discuss and compare the effort that was required to achieve these figures (programming effort and debugging effort). Finally, you should try to explain your findings and try to come up with possible further directions of investigation. Try to be brief; this should not take more than one page.

Task 5: Team

Provide a short description on how you divided up the work, i.e., who did what? Distribute 16 marks amongst all of you to indicate perceived imbalances in work.