UNIVERSITY OF AMSTERDAM
Informatics Institute
Systems and Networking Lab
Parallel Computing Systems Group

Dr.ir. Ana Varbanescu
Dr. Clemens Grelck
Julius Roeder, MSc
Misha Mesarcik, MSc
February 1, 2021

Programming Multi-core and Many-core Systems

— Lab Project Description —

1 Introduction

As a representative example for the parallelisation of numerical problems we study the (much simplified) simulation of heat dissipation on the surface of a cylinder. This problem serves as an example for program parallelisation throughout the course. We begin with the implementation of the sequential base case using plain C. Later on you will parallelise this base implementation using the various technologies taught throughout the course.

Implementing the sequential base case in C can be considered a warm-up exercise to get you on speed with C programming, an essential skill for most parallel programming approaches. After the deadline for this assignment we will provide you with our sequential implementation. It is up to you whether you continue with this or with your own implementation, provided the latter is sufficiently well executed.

This project forms the core of the lab assignments for this course. For each parallelisation concept taught in the course we will add a few smaller assignments that highlight particular characteristics. We provide you with a coding framework that helps you to focus on the interesting parts of each parallelisation concept instead of wasting time writing boiler plate code.

2 Heat dissipation on a cylindrical surface

The cylinder surface shall be represented as a temperature matrix t of $N \times M$ grid points. Each grid point is associated with a temperature value, specified as a double precision floating point number. The M columns of the matrix shall make up the cyclic dimension of the cylinder (i.e. the grid points in the first column are neighbours of those in the last column) and the N rows of the matrix shall represent the other dimension of the cylinder that has fixed boundary conditions. Figure 1 illustrates the transition from the continuous surface of a cylinder to a discrete 2-dimensional grid.

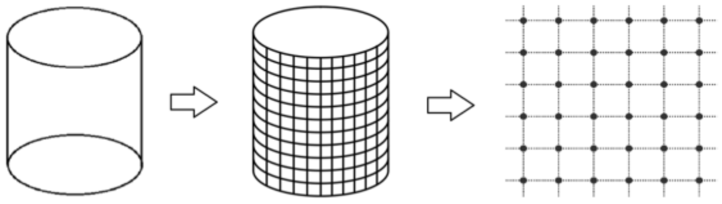

Figure 1: Illustration of cylinder surface discretisation into 2-dimensional grid

The simulation of heat dissipation is an iterative process. At each iteration step the temperature value of each grid point shall be recomputed as the weighted average of the previous iteration's value at that point, the previous iteration's values of the 4 direct neighbour grid points and the previous iteration's values of the 4 diagonal neighbour grid points, depending on thermal conductivity.

Thermal conductivity is a property of the material that makes up the surface of the cylinder. Hence, conductivity coefficients are constant in the time domain, but different in the spatial domain. An $N \times M$ conductivity coefficient matrix c shall contain one (double precision floating point) conductivity coefficient in the interval (0,1) for every simulation grid point. A coefficient of 1 indicates no conductivity whatsoever, i.e. the temperature at the corresponding grid point is not affected by its neighbourhood. In contrast, a coefficient of 0 represents maximum conductivity, i.e. the temperature at the corresponding grid point is not affected by its value in the previous interation. Obviously, neither extreme point of the coefficient value interval is realistic.

Simulation-wise the conductivity coeeficient denotes the weight of the previous iteration's value at the same grid point for the current iteration's value. The joint weight of the direct neighbours shall be $\frac{\sqrt{2}}{\sqrt{2}+1}$ of the remaining weight and those of the diagonal neighbours $\frac{1}{\sqrt{2}+1}$. The weight of each of the four direct neighbours shall be one fourth of their joint weight, likewise for the four diagonal neighbours. Hence, direct neighbours have a stronger impact on the new value than diagonal neighbours, but all direct neighbours have the same impact just as all diagonal neighbours have.

To make a small example, assume a conductivity coeefficient of 0.4. Then, the joint weight of all direct neighbours is 0.3516 and the weight of a single direct neighbour is 0.0879. The joint weight of the four diagonal neighbours is 0.2484 and the weight of a single diagonal neighbour is 0.0621, all with some rounding of real numbers, of course. Figure 2 further illustrates the neighbourhood relation of grid cells.

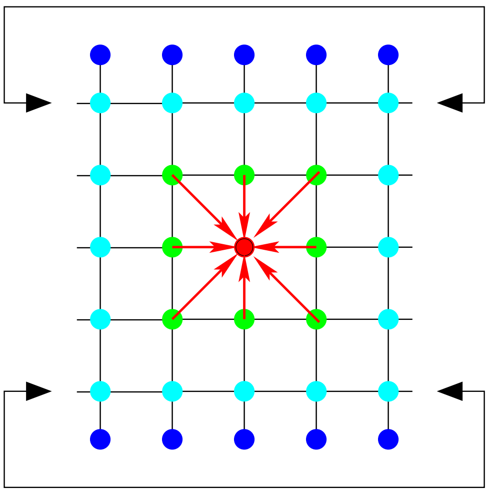

Figure 2: Illustration of fixed value halo rows and cyclic neighbourhood relation in 2-dimensional grid

A particularly interesting aspect of discrete grid representations are what to do with the boundary elements that do not have some of the direct or some of the diagonal neighbours. As illustrated in Figure 2 this is sort of obvious for the rows where the cylinder form demands cyclic neighbour relations. For the columns we work with so-called fixed boundary conditions: each grid point in one of the boundary rows has an associated halo grid point. These halo grid points are initialised with value of the corresponding boundary grid point and remain unmodified throughout the simulation.

The number of iterations is determined in one of two alternative ways. In the first scenario the number of iterations is simply given as a runtime constant $maxiter$. This option is handy for runtime performance experimentation. In the second scenario we check convergence after each iteration and stop once all absolute differences between old temperature value and new temperature value across the entire temperature matrix remain below a given threshold ϵ. This option is more representative for realworld simulations. As convergence may be difficult to predict, we still use $maxiter$ as a cut-off number of iterations to avoid overly time-consuming, and thus computing resource consuming, simulations.

As an additional feature your heat dissipation simulation code shall periodically report a number of characteristic values during the simulation, more precisely after each k iterations as well as after completion of the simulation. Again, k is a runtime constant provided at simulation start. The characteristic values are: the number of iterations processed, the minimum, maximum and average temperature across all grid points as well as the maximum absolute difference between old and new temperature at any grid point as an indicator of convergence.

3 Experimental framework

We provide you with a complete experimental code framework that allows you to focus on the interesting and relevant code sections. You can download the latest version of the framework from Github:

https://github.com/juliusroeder/pmms-heat-diffusion .

The following description serves as a documentation of the user interface of the framework, or, if you want, as a specification of the required behaviour of your code.

3.1 Program command line parameters

The interface to your program should present itself as an executable named `heat` accepting the following command-line parameters:

`-n` N specifies the number of rows;

`-m` M specifies the number of columns;

`-i` $maxiter$ specifies the maximum number of iterations;

`-k` k specifies the reporting period;

`-e` ϵ specifies the convergence threshold;

`-c` $FILE$ specifies the input file to read conductivities from;

`-t` $FILE$ specifies the input file to read initial temperature points from;

`-H` H specifies the highest temperature in the input file;

`-L` L specifies the lowest temperature in the input file;

`-p` P specifies the number of threads to be used (where applicable).

Sample input files are provided in the ASCII Portable Graymap format[1] (P2). This format supports the definition of 2D point grids where each point receives a value between 0 and 255. To translate an input grid to conductivities, the input value 0 should be mapped to conductivity 0 and the value 255 to conductivity 1. To translate an input matrix to temperatures, the input value 0 shall be mapped to the temperature L specified with `-L` and the value 255 to the temperature H specified with `-H`. The mapping shall be linear.

To ease your task and to ensure uniform usability across solutions we provide you with a library of helper functions and a skeleton source tree. You can modify these in any way as long as you keep the base external interface of your program compatible (e.g. names of command-line parameters and input file format).

[1] `http://en.wikipedia.org/wiki/Netpbm_format`

3.2 Program input and parameters

In the provided source file `input.c` we provide a library function with the following interface:

```
void read_parameters(struct parameters *p, int argc, char **argv);
```

This parses the command line parameters and reads the input files. The parameters are then written to the `struct parameters` in the following fields:

- `size_t N, M;` — the matrix sizes N and M;
- `size_t maxiter;` — the maximum number of iterations;
- `size_t period;` — the reporting period in iterations;
- `double threshold;` — the convergence threshold;
- `const double *tinit;` — initial temperatures (values in row-major order);
- `const double *conductivity;` — conductivity values for the cylinder (values in row-major order).

3.3 Reporting of results

After every k iterations and upon completion ($threshold$ or $maxiter$) your program must compute the set of characteristic values as specified above. Your code should afterwards call the provided `report_results` function (`output.c`) with the following interface:

```
void report_results(const struct parameters *p, struct results *r);

struct results {
   size_t niter;    // effective number of iterations performed
   double tmin;     // minimum temperature on the surface
   double tmax;     // maximum temperature on the surface
   double tavg;     // average temperature on the surface
   double maxdiff;  // maximum temperature difference during last iteration
   double time;     // effective computation time in seconds
};
```

The use of this function facilitates evaluation by making the outputs from all implementations homogeneous. You can also modify this code as long as the output format and performance computation remains. Note that updated versions of this code may be provided to you during the course.

3.4 Visualisation

To aid understanding the behaviour of your algorithm you can optionally implement data visualisation during computation. Note, however, that this will not be evaluated. To facilitate this we provide a small library `img.c` to output PGM data with the following interface:

```
void begin_picture(size_t key, size_t width, size_t height,
                   double vmin, double vmax);
void draw_point(size_t x, size_t y, double value);
void end_picture(void);
```

This works as follows: to create an output image with name `key` and size $N \times M$, the program can call `begin_picture(key, N, M, vmin, vmax)`. Then the program calls `draw_point(x, y, v)` for each point in the output to define the computation value v. The final call to `end_picture()` creates the output file.

The parameters `vmin` and `vmax` indicate the dynamic range of values to be represented in the output. All values provided to `draw_point` outside of this range cause saturation in the image file. Note that `begin_picture` and `end_picture` are not thread-safe, i.e. they access shared state without synchronisation.

4 Experimental evaluation

Experiments with your code, reporting on them and interpreting your observations form an integral part of the exercise. In the following we set up a few general guidelines that may be concretised and adapted as needed for the various programming models.

Run each experiment (identical set of simulation parameters) at least three times. Report on the shortest runtime that you achieve, but make sure that this is not due to execution failure, i.e. make sure that all program runs actually execute the required computations. This measure is meant to reduce the impact of coincidental and unrelated events happening on the experimental platform during the execution of your code and likewise to counter the usually observed variance in execution times.

Report problem sizes consisting of the grid size for the cylinder surface and the number of iterations. While in practice it is more common to run simulations until a certain level of convergence is reached, for benchmarking it is usually better to set the convergence threshold such that the given maximum number of iterations effectively determines the actual number of iterations. It remains of course essential that you still compute the convergence level as well as the other simulation figures (i.e. minimum, maximum and average temperatures across the surface) at the specified intervals. For each experimental setting otherwise, report runtimes for *k=1* (very frequent checks) and *k=niter* (only one evaluation after last iteration). What impact do these checks have on the simulation speed that you manage to achieve?

Choose the number of iterations wisely. In practice, simulations like ours run for a very long time and many iterations, but they typically expose the same runtime characteristics in every iteration. For benchmarking we are not really interested in the final result (as long as we actually do the right computations), but in the runtime characteristics. Nonetheless, we need to run a certain number of iterations to reasonably average certain overheads caused by each iteration. We also need to offset one-off setup and finalisation overheads, but for simplicity the framework excludes setup and finalisation stages in the timings.

With different simulation grid sizes the number of iterations becomes the one knob to control how long an experiment effectively takes. Choose it large enough that your results are not blurred by timing inaccuracies and similar. Several seconds is the bare minimum runtime as far as these are concerned. On the other end, a single experiment should normally not exceed a few minutes of compute time on the experimental system.

It is easy to define heat diffusion simulations that utilise the whole DAS-4 cluster for hours. Do not do this! Others want to use the system, too, and the insight gained by such long running experiments is usually not greater than what could be achieved by much shorter experiments. Think before running an experiment! Use your compute time wisely and always consider fair usage of shared resources.

Depending on the specific experiment different figures may be relevant to be reported, but in general absolute runtimes, relative runtimes (e.g. speedups of parallel execution in relation to a sequential base case) and FLOP/s (average number of floating point instructions executed per second) are of interest. The latter can easily be obtained numerically by measuring execution time and computing the total number of floating point instructions executed based on the experimental parameters.

5 Experimental environment

You are encouraged to develop and test your code on your own computer. However, for benchmarking and performance evaluation you must use larger scale parallel hardware. For this purpose we use the DAS-5 cluster of the Advanced School for Computing and Imaging (ASCI), which features a large number of nodes with different numbers of cores plus special nodes equipped with accelerator hardware. Making use of the same hardware also facilitates the comparison of solutions across participants.

Information on the DAS-5 cluster and documentation on how to use it are available at `http://www.cs.vu.nl/das5/`. During the first lab sessions we will also provide a tutorial introduction to using the DAS-5 system.

6 Submission of results

During the course you will submit your source code at different milestones (deadlines) in the form of a compressed archive. The submission archive can be created by the following command:

```
make submission_N
```

where N is in the range 1–4. This archive includes the additional assignments beyond the heat diffusion project. Submit the resulting archive file through Canvas before the deadline (midnight).

Make sure to execute everything on the DAS-5 !!

Submit your work in fixed groups of two. Fixed means that once chosen you stick to your partner for the rest of the course, unusual circumstances apart. Group work means that you work on all assignments *together*. Group work does *not* mean that you split the assignments among the two of you. If we get the impression that the workload and/or knowledge is too unevenly split between partners, we reserve the right to conduct individual examinations on the subject of the assignment and the course topics in general.

7 Hints on report writing

Your report is supposed to explain your solution to someone who is familiar with programming, in our case in C, including the parallel programming paradigms used throughout the course. Briefly summarize the assignment as an introduction. Normally defining the research question would be major part of such a report, but as this is given as an assignment, do not waste time to rephrase the assignment in all details.

Then describe your solution at a high level of abstraction, i.e. do not copy the whole program code into the report. It is, however, often relevant and interesting to copy some lines of code that either contain the key solution to the given problem or that you find for whatever reason interesting to talk about. If needed such code snippets can also be simplified and beautified to improve readability.

In this part of the report you should also describe why you came up with your solution, what alternatives you considered and rejected for what reasons, etc. Anything you find remarkable or super smart about your solution should be elaborated on here. Convince the reader (us) that your solution is the best thing since sliced bread.

If you feel that your solution is not the best thing since sliced bread, also discuss that. Explain potential shortcomings and failures; explain why you could not solve them (e.g. submission deadline was 5 minutes ahead when you figured it out) and sketch out what you think would be a way forward. This can be the basis for a good report, even if the programming exercise did not work out that well for you.

Various assignments ask you to run certain experiments. Reporting on the outcome of these experiments is a critical part of your report. Whenever possible, present your findings in a graphical way and discuss them in the text. If you feel more or different experiments could be interesting, run them as well and report on them.

Try to draw conclusions from the experiments. Why does this code perform better than that code. Why does this code scale to 4 cores, but not to 8, etc. All these are interesting and relevant questions as well as an opportunity for you to demonstrate your knowledge.

Please, use the LaTeX style provided in the git repository; the page limit is 10 pages in this style. Should you have a dire need to submit more plots, graphs, etc, put them into a clearly marked appendix.

By now it should be crystal clear that report writing is a significant part of each assignment. Take this into account when planning your time.

Your job is not done when your code compiles without errors!