
UNIVERSITY OF AMSTERDAM

Informatics Institute
Systems and Networking Lab
Parallel Computing Systems Group

Andy Pimentel
Simon Polstra
Acknowledgement:
The heat dissipation project was created by:
Ana Varbanescu, Clemens Grelck, Jelle van Dijk,
Misha Mesarcik

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. We begin with the implementation of the sequential base case using plain C. You will then parallelise this sequential implementation using OpenMP. And finally you will extract the memory trace from the parallel version and analyse the cache behaviour with your cache models from assignment 2 and 3.

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 Submission & Assessment

For assessment you must:

1. Write a report with your results, explanation of the results and a comparison between the experiments that you performed.
2. Submit your source code and report packed in `tar/gz` format. You can run `make` from the root directory of the project to create the tar file `heat_submit.tar.gz` that contains all your source files and your report.
3. If we think it is necessary we can ask you to demonstrate your solutions and code during one of the lab sessions.

Submission Requirements:

1. Your sequential and OpenMP code should compile cleanly without warnings on a recent mainstream Linux distribution without any modifications to the submitted code, either using the provided Makefile or your own, which has to be included.
2. Do not change program command line parameters as explained in section 6.1.
3. Do not change how the results are reported as explained in section 6.3
4. Your program should always terminate and exit without any errors.
5. Your submission should contain a report named `report.pdf` in which you describe the problem, the sequential and parallel implementations, discuss and analyse your findings and results.

3 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.

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 iteration. Obviously, neither extreme point of the coefficient value interval is realistic.

Simulation-wise the conductivity coefficient 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 coefficient of 0.4. Then, the joint weight of all direct neighbours is 0.3515 and the weight of a single direct neighbour is 0.0879. The joint weight of the four diagonal neighbours is 0.2485 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.

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

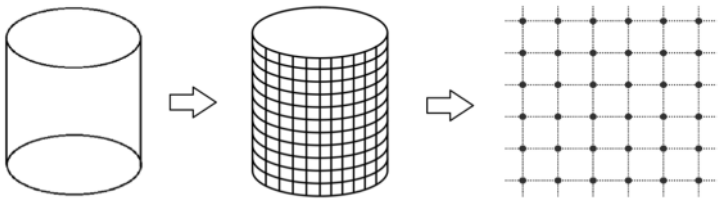

Figure 1: Illustration of cylinder surface discretisation into 2-dimensional grid

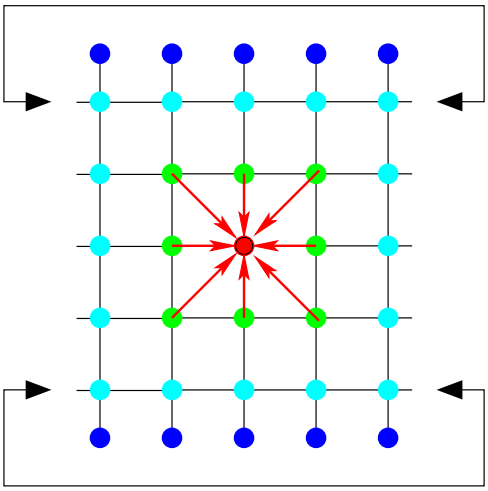

Figure 2: Illustration of fixed value halo rows and cyclic neighbourhood relation in 2-dimensional grid

entire temperature matrix remain below a given threshold ϵ. This option is more representative for real-world 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.

4 Assignment

The assignment consists of four parts. The first part is to implement the sequential version of the heat dissipation problem. The second part is to parallelise the sequential implementation using OpenMP. After you have finished the parallel implementation you need to generate memory traces from by running it with a tracing library. The final step is to simulate the memory trace in your cache simulator and analysis the results. This can be an iterative process where the analysis of the simulation results can help you to improve your parallel implementation.

4.1 Sequential implementation

The first step is to implement a sequential solution for the heat dissipation problem. You do not need to measure the performance or obtain the address trace of this program. The sequential implementation will serve as a starting point for the parallel implementation, so make sure it is functionally correct before you start to parallelise it. We provide a framework for the assignment which is explained below. The framework code for this assignment is located in the `heat_seq` directory of the framework.

4.2 Parallel implementation

The next step is to parallelise your sequential implementation. OpenMP is designed to make this as easy as possible. Depending on your sequential implementation it can be as simple as copying your code to the `heat_omp` directory and adding OpenMP pragmas. You can start with the simplest approach to split your program into multiple threads. You can later try different parallelisation and thread scheduling approaches to improve the cache behaviour of your heat dissipation program.

4.3 Address tracing

To analyse the cache behaviour of your parallel heat dissipation program you first need to generate an address trace that captures the memory access patterns of the heat dissipation computation. For this we provide a library that adds tracing code to your heat dissipation binary at runtime. When the heat dissipation program is run with this tracing library the added tracing code generates a text file for each thread that contains all the memory operations of that thread in the program. By default the traces of different processors in the `psa` are not synchronized, so it does not make sense to trace more than one iteration of the heat dissipation program. You can set the iteration count to one with the command line parameter `-i 1` as explained below. The make target `run_with_tracing` will do this automatically. You can however add a `barrier` annotation to your program that synchronizes the tracing in the `psa` library. With such a barrier you can trace more than one iteration of the heat dissipation in a realistic manner.

4.4 Simulation and analysis

Before you can simulate the address trace in your own cache simulator you will first need to perform some post processing. The trace file text generated by the trace library needs to be checked and converted to `trf` tracing binary format of the `psa` library. This is explained below. With the trace now in the binary `trf` format you can use it to perform experiments with your cache simulators from assignment 2 and 3. You can experiment with different cache configurations and heat dissipation parameters to see how they can affect the performance.

The final step is to analyse the results of the experiments. Can you explain the observed behaviour considering the way you parallelised the application? Can you think of ways to improve the way you parallelized the application? Consider the loop structure and scheduling of the threads here. With these new insights you can now change your parallel implementation, rerun the trace generation and simulate the new traces.

5 Address tracing

The tools we use to create address traces only work on `x86` architectures. If you don't have access to such a machine you can use the DAS-5 cluster to generate your address traces. Trace generation on the DAS-5 cluster is discussed in the next section.

To generate your own address traces you need to install the Intel Pin software and an address trace library called `mcps-tracer`. This tracing library uses the Pin software to dynamically instrument the binary so that all the memory accesses are traced to a text file.

You will first need to install the Intel Pin software. Download the pin tar file from the OpenMP assignment page on Canvas. For the installation you only need to untar the archive. In the example below we install the Pin software in the `~/projects` directory and add a symbolic link to refer to the Pin installation as `~/projects/pin`:

```
~/projects$ tar xf ~/Downloads/pin-3.27-98718-gbeaa5d51e-gcc-linux.tar.gz
~/projects$ ln -s pin-3.27-98718-gbeaa5d51e-gcc-linux/ pin
```

The next step is to install the tracing software. Download the tar file `mcps-tracer.tar.tgz` from the OpenMP assignment on Canvas. The address tracer uses Intel Pin, so we add the Intel Pin installation directory to `PATH` and set it in `PIN_ROOT`:

```
~/projects$ export PATH+=:/path/to/pintool/pin
~/projects$ export PIN_ROOT=/path/to/pintool/pin
~/projects$ tar xf ~/Downloads/mcps-tracer.tar.gz
~/projects$ cd mcps-tracer
~/projects/mcps-tracer$ make
```

If all went well we now have a shared library `tracer.so` in the `tool/obj-intel64` directory.

With the tracing tools installed we can now test if the tracing works. Download the experimental framework archive `heat-diffusion.tgz` from the OpenMP assignment and unpack it and enter the `trace_example` directory.

```
~/projects$ tar xf ~/Downloads/heat-diffusion.tgz
~/projects$ cd heat-diffusion
~/projects/heat-diffusion$ cd trace_example
```

The `trace_example` directory contains the following simple OpenMP program called `simple_add_example.c` that shows how to trace a specific part of a program:

```
int main(void) {
    int sum = 0;
    #pragma omp parallel num_threads(2)
    {
        printf("thread: %d\n", omp_get_thread_num());
        #pragma omp for
        for (int i = 0; i < 4; i++) {
            start_roi();
            sum += 1; // race condition!
            end_roi();
        }
    }
    printf("sum: %d\n", sum);
    return 0;
}
```

Ignoring the race condition for now we see that the program creates two threads that will each execute two iterations of the for loop. Often you only want to trace a specific part of program, this is referred to as the *region of interest* or *roi* for short. We start such a region with the `start_roi()` marker and end it with the `end_roi()` marker. You can have multiple regions of interest in your code. In this case we are only interested in the address trace of the body of the `for` loop so we place the markers inside the loop body. If we would place the roi marker around the whole `for` loop the tracing would include the overhead of the setup of the parallel `for` loop. And if we would add an `omp critical` pragma to fix the race this would then also be included in the address trace. It is important to note that a thread will only start and stop tracing when it *itself* encounters a roi marker. Threads do not inherit the tracing status from the master thread. So if we would place the `start_roi()` marker at the beginning of `main` before the start of the parallel region, we would only trace the master thread, since it is now the only thread that actually executed the start marker. You can decide yourself how much loop and OpenMP overhead you want to include when tracing your own heat dissipation program, but be sure to only set the markers inside your parallel sections otherwise the helper threads are not traced.

Before we can trace this example program we need to make sure that the mcps tracing library can be found. Set the `MCPS_TRACER_ROOT` in the Makefile of the example program to the installation location of the tracer. In our case that is `~project/mcps-tracer`. Also make sure that the environment variable `PIN_ROOT` that we set above with the `export` command still points to Intel Pin installation directory. In our case that is `~/projects/pin`. We can now compile the example program and generate the address trace with the following commands:

```
trace_example$ make
trace_example$ make run_with_tracing
```

Running the `make run_with_tracing` command executes the binary together with the tracing library to generate the address trace of the region of interest code. The memory tracer generates ascii trace files, one for every thread in the program. The thread id is added as a suffix. The default output file name is `address_log`. Since the `simple_add_example` has two threads the tracer generates two files: `address_log.00` and `address_log.01`. You can specify a different output basename with the `-o` flag.

The final step to create a memory trace that we can use in our cache simulator is to convert it to the binary TRF format used in the PSA trace reader library:

```
trace_example$ make memory_trace.trf
trace_example$ ../scripts/trace_printer.py memory_trace.trf
5TRF 2
P0 WRITE 140723217582368
P1 WRITE 139640547831152
```

```
P0 READ 140723217582368
P1 READ 139640547831152
P0 READ 140723217582376
P1 READ 139640547831160
P0 READ 140723217582468
P1 READ 140723217582468
P0 WRITE 140723217582468
P1 WRITE 140723217582468
...
```

The `make memory_trace.trf` command above uses the `trace_converter.py` script in the `scripts` directory to combine all the ascii traces from the threads in the traced program into a single binary TRF trace file. Since we assume all memory operations are 4 bytes in our cache model the trace converter filters the trace to make sure all accesses are 4 bytes aligned by default. Aligned accesses are usually much faster so it is unlikely that many trace events will be filtered in this step.

When we print the trace we can see that each thread accesses data in their own stack frame and we also see that both threads update the shared `sum` variable at address 140723217582468. We now have a trace that we can run in our cache simulator.

The compiler, compiler settings and machine can all have a big effect on the generated traces. It could be interesting to compare the effect of the compiler optimisation level on the memory access patterns. But do make sure to compare traces generated with the same machine unless you are specifically interested in the effect the different systems have on the traces.

5.1 Address tracing on DAS-5

We have installed the tools to create memory traces on the DAS-5 cluster. If you want to use the DAS-5 for this assignment you can request an account from me via email (s.polstra@uva.nl). With these credentials you can login to the head node `fs2.das5.science.uva.nl` with `ssh`. Pin and the mcps tracing library are installed in `/var/scratch/spolstra/mcps`. This directory also contains the heat-diffusion archive and example job scripts that you can use to compile and run programs on a DAS node. **Do not run heavy jobs on the head node itself or your account will be suspended**. Always start heat dissipation runs and trace generation with the `sbatch` command so they run on a DAS compute node.

After we unpacked the heat-diffusion archive in our home directory on the DAS head node and changed the `PIN_ROOT` and `MCPS_TRACER_ROOT` variables in the makefile to point the tools in `/var/scratch/spolstra/mc` we can run the small example trace:

```
~/trace_example$ grep ROOT= Makefile # should point to /var/scratch/spolstra/..
~/trace_example$ cp /var/scratch/spolstra/mcps/simple_add_example.job .
~/trace_example$ sbatch simple_add_example.job
~/trace_example$ make memory_trace.trf # Always use the job script for larger traces!
```

To generate address traces for your parallel heat application you can simply change the `APP_DIR` variable in the job script. The directory `/var/scratch/spolstra/mcps` contains example sbatch job scripts that you can use for the sequential and parallel heat dissipation project.

We have installed systemc `/var/scratch/spolstra/systemc-3.0.1/install` so you can also run your cache simulations on the DAS-5 cluster if you want to. You will need to set the `SYSTEMC_PATH` variable in your Makefile to this directory. More information on the DAS-5 cluster and documentation on how to use it is available at `http://www.cs.vu.nl/das5/` and in the DAS tutorial document on Canvas.

6 Heat dissipation framework

The `heat-diffusion.tgz` archive contains a complete experimental code framework for the heat dissipation project that allows you to focus on the interesting and relevant code sections. You can download the latest version of the framework from Canvas. In this framework you will only need to modify the `compute.c` file in the `heat_seq` and `heat_omp` directory.

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.

6.1 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 Gray map 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).

6.2 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).

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

6.3 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 allows you to check if the simulation is functionally correct by making the outputs from all implementations homogeneous. You can also modify this code as long as the output format and performance computation remains.

6.4 Visualisation

Although it is not necessary for this assignment you can 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.

7 Experimental evaluation

You only need to run multiple iterations of the heat dissipation calculation to test the correctness of sequential and parallel implementations. Run the program without address tracing for these tests. Test that the output matches the reference output in the `reference_output` directory and that the programs stops when the convergence threshold is reached.

To generate a memory trace from the parallel heat dissipation program you should run one iteration of the heat dissipation calculation for the reasons mentioned in section 4.3. Experiments with your parallel implementation, reporting and interpreting your observations form an integral part of the assignment. In the previous assignments the memory traces were fixed and you could only experiment with the cache configurations. In this assignment you know the structure of the program. You can now analyse the access patterns in the simulator and then modify your application to improve its cache performance. This is an iterative process that can give you more insights into the access patterns of the code and how they can effect the performance of the cache system. You can also create different sized input with the `Makefile` in the `images` directory to see how that effects the cache behaviour.

Run your traces on your valid-invalid multiprocessor cache system and on your MOESI multiprocessor cache system and compare the results.

8 Hints on report writing

Your report is supposed to explain your solution to someone who is familiar with programming, in our case in C and OpenMP.

Describe your solution at a high level of abstraction, i.e. do not copy the whole program code into the report. It is, however, sometimes 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.

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. If needed also 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.

Reporting on the outcome of these experiments is the *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 and report on them as well.

Try to draw conclusions from the experiments. How and why does a different loop parallelisation method change the hit rates. Does the MOESI cache system improve performance and if so what are the main benefits. How does the OMP thread scheduling effect the memory access pattern. How does scaling to more threads effect the cache behaviour. All these are interesting and relevant questions as well as an opportunity for you to demonstrate your knowledge.