

## Lab 4: Heat Diffusion

**Assigned:** 23 March 2001 **Due:** 11 April 2001, 11:59pm **Lead Person:** Umut A. Acar

### 1 Logistics

You can retrieve your files with the `get` command. This assignment has two parts and you will hand in two files, `heatLocal.c` and `heatFast.c` for the first and the second part respectively. You can hand in as many times as you want. For the second part, you need to submit (i.e., run `make submit`) to obtain an accurate performance measurement. The command `make submit` submits your files for performance evaluation on a machine dedicated for this

Please follow the web page and the billboard for announcements on this assignment.

### 2 Introduction

We understand that over the spring break, you would like to talk to folks at home about the cool work that you did this semester. “Defusing bombs” or “collecting garbage” just does not sound quite right for that. So in this lab, you will simulate “heat diffusion with the Jacobi Iteration technique.”

Consider a cold iron bar and imagine that we heat the bar at two ends. When would the middle of the bar be warm, or how warm would it be after say 10 seconds? One way to determine the answers is to actually do the experiment, and the other is to simulate the experiment. In this lab, you will learn some techniques to efficiently perform such a simulation.

In your simulation, you represent the metal bar with an array of structures of type `cell`. A cell is defined as follows.

```
// The cell structure
typedef struct {
    float temp;           // The temperature.
    float alpha;         // The radioactivity measures.
    float beta;
    float gamma;
    double eta;
    double zeta;
} cell;
```

You can imagine each cell in the array representing of one inch of the bar in order. So the first cell represents the first inch the second the second inch and so on. In your simulation, you are interested in one member of the `cell`, `temp`.

Given an array of cells representing the metal bar, you will perform the simulation in so called steps. In each step, you will update the temperature of a cell by the average of the temperatures of its immediate neighbors, including itself. For example, you assign the temperature of the cell in

the middle of the array to the “sum of the neighboring left and right elements and itself divided by 3”. Each step simulates the heat diffusion over a small time period, say one tenth of a second. This is a sample “Jacobi Iteration”. Each iteration changes the temperature of each element “locally”. For example, if you imagine a cold bar heated at two ends, the middle of the bar stays cool until later iterations.

### 3 Your Task

The function `heat` below performs the simulation for `N_STEPS` steps.

```
// The naive implementation
// Input: The length of the grid and the grid.
cell* heat (int n, cell* grid)
{
    int step;
    int i;

    // Compute the result.
    for (step = 0; step < N_STEPS; ++step) {
        grid[0].temp = (grid[0].temp+grid[1].temp)*0.5;

        for (i = 1; i < n-1; ++i)
            grid[i].temp = (grid[i-1].temp+grid[i].temp+grid[i+1].temp)*0.33;

        grid[n-1].temp = (grid[n-2].temp+grid[n-1].temp)*0.5;
    }

    return grid;
}
```

The for loop goes through the array, `grid`, for `N_STEPS` steps, which I set to 20 in the `defs.h`. At each step, each element is updated to the average of itself and its neighbors.

Unfortunately, this simple implementation delivers poor performance due to poor data locality. If the array `grid` does not fit into the cache, then the `grid` is loaded into the cache at each iteration.

Your assignment has two parts. In the first part, you optimize the code given above for better cache performance. Your aim is to minimize the number of cache misses. When working towards this goal, your techniques may increase the work (the number of executed instructions) compared to the naive `heat` implementation above. All your work for this part should be in file `heatLocal.c`.

In the second part, you optimize the naive implementation for performance. You will measure the performance with throughput; the higher the throughput the better the performance. I defined the throughput of heat diffusion as the total number of floating point operations executed divided by the number of seconds that their execution takes. In the first part, you decrease the number of cache misses by possibly doing more work - this is typical of most cache efficient implementations. In this part, we would like you to pay more attention to the additional work performed as you decrease the number of cache misses. All your work for this part should be in the file `heatFast.c`.

Since you are sharing machines with others, your performance measurement for the second part may not be accurate. To obtain an accurate performance measurement submit your program to

the class server with `make submit`.

## 4 Mechanics

When you run the `get` command, you will retrieve the necessary files for this assignment. There is a driver program, `driver.c`, which you can use for testing. The driver program takes the length of the grid as a parameter and allocates a grid (an array of cells) of the given length and sets the `temp` of each cell to a randomly generated floating point number. Once the grid is initialized the driver program tests `heatLocal` and `heatFast`. The driver calls the function `heatLocal` on the grid and verifies that `heatLocal` returns the correct result. Then the driver calls `heatFast`, verifies that it returns the correct result and prints the execution time and MFLOPS.

The functions `heatLocal` and `heatFast` are implemented in the files `heatLocal.c`, and `heatFast.c` respectively. By default, both functions perform the simple `heat` implementation described in Section 3. You should edit the two files `heatLocal.c` and `heatFast.c` and replace the given implementations of the function `heatLocal` and `heatFast` with your implementation.

## 5 The Cache Profiler `cacheprof`

To count the number of cache misses that your program incurs, we use an open-source software, `cacheprof`. You can learn more about `cacheprof` at <http://www.cacheprof.org>.

I set up `cacheprof` to simulate your program with a cache that is equivalent to the L1 cache on the fish machines. The L1 cache of a fish machine is a direct mapped, 4-way associative, 16KB cache with 32-byte lines. When you run your program, `cacheprof` counts the number of cache misses incurred by simulating your program with the above cache. Since this is a simulation, the number of cache misses that `cacheprof` finds does not depend on the particulars of the system used or the load on the system. Thus, for the first part of your assignment, you do not need to submit your program to obtain an accurate measurement.

Using `cacheprof`, you can instrument a particular file by preceding the compilation command that generates the object code by the command `cacheprof`. This is what we do to instrument `heatLocal.o` and `heatFast.o` (see the `Makefile`).

When your program terminates `cacheprof` shows you a number of performance metrics. Here is an example output by `cacheprof`.

```
==cacheprof== level-2 instrumented program: run complete
==cacheprof==      200,101 insns
==cacheprof==      80,032 refs  (      60,008 rd +      20,024 wr)
==cacheprof==      20,043 misses (      20,021 rd +          22 wr)
==cacheprof== 0.02 seconds, 10.01 MIPS
```

The information above pertains to the instrumented parts of the executed program. It tells you the number of instructions, the number of memory references, and their composition, the number of cache misses, and the execution time and the MIPS. You should ignore the last line which contains the timing and the MIPS and evaluate your program with respect to the information printed out by the driver.

Since the output of `cacheprof` pertains to the whole program, you should profile either `heatLocal` or `heatFast` at a time. For your convenience, I have defined two flags in `defs.h` that enables you to work on either part independently (see the file `defs.h` for how to use them).

## 6 Compiling, Debugging, Working with the Makefile

There are two different compilation modes that you can use for compilation. In the default mode, all code is compiled with optimization and `heatLocal.c` is profiled by `cacheprof`. The second mode is for debugging. To switch between the two modes you edit the `Makefile`. Once you made the appropriate changes to `Makefile`, type `make` to compile your program.

To switch to debugging mode, you need to turn `cacheprof` off (`cacheprof` confuses debuggers). First, set the variable `CFLAGS` for debugging (`-g` option). Second, change the compilation command for the targets `heatLocal.o` and `heatFast.o` by commenting out the corresponding lines. This will enable the default compilation for `.c` files in the `Makefile`, which is performed without profiling.

To switch back to default mode, undo the above changes. Also, if you would like to see the cache performance of your `heatFast.c` during the second part, uncomment the two lines for the target `heatFast.o`. Since `cacheprof` slows your programs down, you will not obtain accurate throughput measurements when you compile `heatFast.c` along with `cacheprof`.

Every time you change the `Makefile`, you should run `make clean` before you run `make`. Also, you run `make` twice after a `make clean`.

When you are working on a part of the assignment, I recommend that you adjust the flags in `defs.h` accordingly. This saves you time and also may prevent possible confusion. Please see the file `defs.h` for how to do this.

## 7 Grading

We will grade your program by running it with several different grid lengths covering a wide range of sizes. We will assign a score to each run based on a scheme that we will announce later on the web page.

To receive points in this assignment, your program should pass the correctness test. The correctness test, checks whether the result returned by your algorithm matches the result computed by the naive algorithm described in Section 3. The test, however, ignores the first and the last `N_STEPS` cells in the result grid. This should help simplify your implementation significantly.

## 8 Make Handin and Make Submit

Before you hand in, make sure your program passes the correctness tests and does not print out any extraneous information. To hand your program in type `make handin`.

To obtain an accurate performance measure submit your program to the server by typing `make submit`.