15-213, Spring 2003
Lab Assignment L4: Simulating Heat Diffusion
Assigned: February 18, Due: Wednesday, March 5, 11:59PM

Shaheen Gandhi (`sgandhi@andrew.cmu.edu`) is the lead person for this lab.

1 Logistics

You may work on this assignment in teams of two, or individually if you wish. To obtain the files for this assignment, copy the following file to your `213hw/L4` directory:

`/afs/cs/academic/class/15213-s03/labs/L4/L4.tar`

You can extract the archive with the following command:

`tar xvf L4.tar`

The archive contains quite a few files. Many of them are helper routines related to performance evaluation. You will be concerned with `heat.c`. It is where you will implement your well-performing code for the task described in this document.

To build the executable, just type `make`. This will build the test application along with the file `heat.c`. We have provided the naive solution to the task in `heat.c` to start you off.

To run the test application, you can run `./heat_test` n and `./cachegrind ./heat_test -c` n. n is just an integer that dictates the size of each dimension of the grid (explained below). We will be grading against $n = 64, 128, 256, 512, 1024$.

Running `./heat_test` n will provide you with a CPE (Cycles Per Element) number. This is explained more below.

Running `./cachegrind ./heat_test -c` n will invoke the test application through `cachegrind`. `cachegrind` is an application that profiles the

cache usage of another application. Note that the "-c" argument is important - it will make `heat_test` run `heatFast` only once rather than multiple times. We will use the number of D1 misses as an indication of performance along with the CPE number. More about how your solution is graded is at the end of this handout.

`cachegrind` outputs a file called `cachegrind.out` every time you run it. This file contains a lot more statistics than what `cachegrind` outputs by default. You can see exactly how many cache misses your algorithm incurs by running `./vg_annotate heat_test`. You can even see line-by-line statistics by running `./vg_annotate heat.c`.

2 Introduction

In this lab, you will simulate "Heat Diffusion with Jacobi Iteration." Consider a cold iron plate and imagine that we heat the plate at the center. When would the corners be warm? One way to determine the answer is to do the experiment, and another is to simulate it.

In the simulation, you represent the metal plate with a 2D array of doubles. You can imagine each element of the array representing the temperature of one square inch of the bar in left-to-right, top-to-bottom order.

Given an array of doubles representing the metal plate, you will perform the simulation in so called steps. In each step, you will update the temperature of an element by the average of the temperatures of its immediate neighbours, including itself. For example, a value in the middle would be updated by its 8 neighbors as well as itself. 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, the corners of a plate heated at the center will stay the same temperature for some time.

You can imagine the cases for the corners can make the code hairy to look at. Therefore, we are providing a boundary of cells that you need not touch (these would be called boundary conditions in a real simulation). Your code will not have to compute averages for the rows indexed by 0 and n-1, nor the columns 0 and n-1. If you picture the grid, you can see a border of cells around the cells that your code changes.

For example, if n = 64, you will only have to perform the simulation for cells in rows 1 through 62 and columns 1 through 62.

3 The Task

The function heat below performs the simulation for N_STEPSsteps.

```
// ith row, jth column
#define RIDX(i,j,n) ((i)*(n) + (j))

double* heat(int n, double* grid)
{
  int step;
  int i, j;

  // Compute the result
  for (step = 0; step < N_STEPS; ++step) {
    for(i = 1; i < n-1; ++i) {
      for(j = 1; j < n-1; ++j) {
        grid[RIDX(i,j,n)] = (1/9.0) * (
                            grid[RIDX(i-1,j-1,n)] +
                            grid[RIDX(i-1,j,n)] +
                            grid[RIDX(i-1,j+1,n)] +
                            grid[RIDX(i,j-1,n)] +
                            grid[RIDX(i,j,n)] +
                            grid[RIDX(i,j+1,n)] +
                            grid[RIDX(i+1,j-1,n)] +
                            grid[RIDX(i+1,j,n)] +
                            grid[RIDX(i+1,j+1,n)]);
      }
    }
  }

  return grid;
}
```

The for loop goes through the array, grid, for N_STEPS steps, which is set to 10 in the heat.h. At each step, each element is updated to the average of itself and its neigbours.

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 is to increase the performance of the task above as much as possible. Suggested courses of action include optimize cacheing characteristics as well as finding faster ways to compute the average.

The performance of your code is measured in Cycles Per Element (CPE). heat_test will test your code against a data set of the size you specify. Once it has verified the correctness of your solution, it will run and benchmark your solution. The total number of cycles taken by your solution will be divided by the size of the data set. This is the number that is reported as Cycles Per Element.

Also, when grading your solution, we will take into account the number of cache misses of your algorithm. You can find the number of cache misses of your solution by running `cachegrind` mentioned above.

Since you are sharing machines with others, your performance numbers may not be accurate. To obtain an accurate performance measurement, submit your program to the class server as described below.

4 Mechanics

Of the files included in `L4.tar`, you only need to edit heat.c. Edit the function `heat` with your solution. I recommend keeping backups of possible solutions while working on new ones by renaming the function and making a new copy in the file heat.c.

You can submit your solution to the class web server to be evaluated. Go to the labs web page (`http://www.cs.cmu.edu/ 213/labs.html`). Under L4, you will find links to register a group, submit your solution, and see where you are in relation to other groups in the class.

5 Grading

We will grade your program by running it with several different data set sizes (64, 128, 256, 512, and 1024) covering a wide range of inputs. We will assign a score based upon the performance of your solution on the data sets.

All cases will be tested for throughput (CPE score). The 256 case will be

tested for cache performance. The average throughput of all cases, along with the D1 cache misses of the 256 case will be used to determine a final score. This final score will determine your grade. Hint: your code can make 256 a special case and perform a different algorithm than your regular solution.

You will not receive any points for programs that do not pass the correctness test. The correctness test will be performed each time you run `heat_test` (without the "-c" argument). Your solution does not have to perfectly mimic the behavior of the naive solution above. However, each element in the final array can have an error of, at most, `EPSILON`. This is defined in heat.c.

6 Handin and Update

Make sure your program passes the correctness tests and does not print out any extraneous information before you hand in. To submit your solution to the class web server, go to the class web site and follow the link under this lab to go to the submission page. From there, you can select a solution file to handin. The latest solution handed in will be considered your final solution. Submitting solutions past the due date will count against late days.