Programming Project 04
Main Memory Performance Modeling
CS 441/541 – Spring 2014

Due Dates & Grading

This project is worth 50 points total. Grades will be assessed as follows:

Project Available		April 7
Component	**Points**	**Due Date**
Project 4		April 21
hwloc component	5	
Scalar Multiplication	15	
Matrix Multiplication	25	
Style & Documentation	5	
(Bonus) Performance Boosters		
Scalar Multiplication	+5	
Matrix Multiplication	+5	
Project Total	50	

Late assignments will be subject to the late policy in the syllabus. Your assignment must be written in C (not C++). No credit will be given for assignments written in any programming language other than C.

Your programs must be able to be compiled using a Makefile by typing the command `make` without any additional arguments in the assignment directory. The assignment directory should contain all of the source files for all of the project components, and a single `make` should compile all of the components.

All parts of this assignment will be completed **individually**.

Overview

In this assignment you will be experimentally exploring the memory hierarchy using two scientific application kernels.

This project is divided into multiple components. Make sure you complete all of the components.

A skeleton template is provided on D2L to get you started. **If you are running on a Linux machine**, you will need to edit the `Makefile` to set the appropriate `LDFLAGS` for your system. See comments in the `Makefile` for more details.

Warning: Running the experiments for large matrix values will take a long time! On the order of a few hours for matrix multiplication.

Project 4.1 – Access the Memory Hierarchy using hwloc – (5 Points)

In this component of the project you will need to install and use the hwloc library to access a representation of the memory hierarchy on your testing machine.

We will be using the hwloc `lstopo` and `lstopo-no-graphics` commands to generate a representation of your machine's topology. This can be the textual representation or the visual representation or both.

You will **need to include this** topology information in your writeup for this project.

Download the hwloc library from the following website:
`http://www.open-mpi.org/projects/hwloc/`

Follow the instructions below to install hwloc in your home directory:

1. Create a location to install hwloc

```
shell$ cd $HOME
shell$ mkdir local
shell$
```

2. Configure the library (the `prefix` argument sets the installation directory, which you can customize to any location you would prefer)

```
shell$ tar -zxf hwloc-1.9.tar.gz
shell$ cd hwloc-1.9
shell$ ./configure --prefix=$HOME/local
shell$
```

3. Make the library

```
shell$ make
shell$
```

4. Install the library

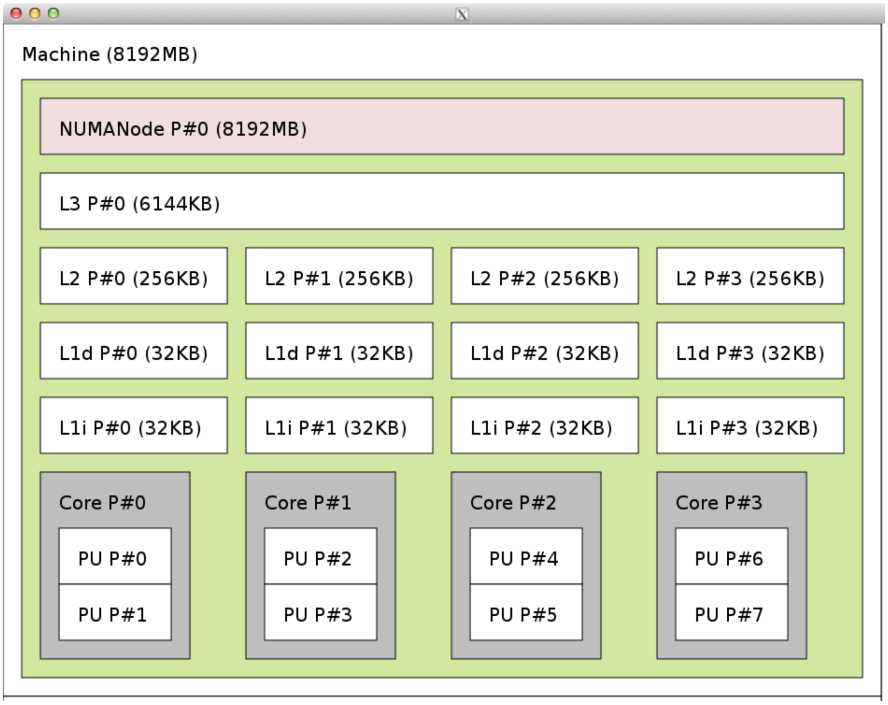
```
shell$ make install
shell$
```

5. Setup environment variables (you need to do this step every time you want to run hwloc). If you modified the prefix variable during the configure phase, then you need to update the paths below as appropriate.

```
shell$ export PATH=$PATH:$HOME/local/bin
shell$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/local/lib
shell$ export MANPATH=$MANPATH:$HOME/local/share/man
shell$
```

6. Run the `lstopo-no-graphics` command. The `lstopo` command can be used to generate a graphical representation of the physical topology.

```
shell$ lstopo-no-graphics
Machine (8192MB) + NUMANode L#0 (P#0 8192MB) + L3 L#0 (6144KB)
  L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
    PU L#0 (P#0)
    PU L#1 (P#1)
  L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
    PU L#2 (P#2)
    PU L#3 (P#3)
  L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
    PU L#4 (P#4)
    PU L#5 (P#5)
  L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
    PU L#6 (P#6)
    PU L#7 (P#7)
shell$ lstopo
```



Project 4.2 – Scalar Multiplication – (15 Points)

In this component of the assignment you will be implementing a simple scalar multiplication program in **two different ways**. This program will generate a random matrix (by calling the allocate_matrix and clear_matrix functions in the support library) and multiply each element of that matrix by a single scalar value. The scalar must be a randomly generated, positive number that is neither 0 nor a multiple of 2. The first variation of the program will implement the following code.

```
1  double run_experiment_ij(mtype_t *matrix,
2                           mtype_t scalar, int N) {
3     for(i = 0; i < N; ++i ) {
4        for(j = 0; j < N; ++j ) {
5           matrix[i][j] = scalar * matrix[i][j];
6        }
7     }
8  }
```

The second variation of the program will **flip the two nested loops**, as seen below. Notice that the only difference is the loop ordering (and function name).

```
1  double run_experiment_ji(mtype_t *matrix,
2                           mtype_t scalar, int N) {
3     for(j = 0; j < N; ++j ) {
4        for(i = 0; i < N; ++i ) {
5           matrix[i][j] = scalar * matrix[i][j];
6        }
7     }
8  }
```

Timing Notes

The matrix will be of type `mtype_t` which is defined in the support library. Since we cannot allocate contiguous two-dimensional arrays in C, we need to impersonate a two-dimensional array using a one-dimensional array. To do so you should use the allocate_matrix function in the support library to allocate a square matrix of the appropriate size. You can also use the clear_matrix function in the support library to initialize a square matrix. You should use the GET_INDEX macro in the support library to calculate the appropriate offset into the one-dimensional array. For example:

```
1  mtype_t *matrix = NULL;
2  // Allocate a matrix of size: N x N
3  allocate_matrix(&matrix, N);
4
5  // Clear/Initialize the matrix
6  clear_matrix(matrix, N);
7
8  // Access matrix[i][j] in the 1D array
9  matrix[ GET_INDEX(i, j, N) ] = ...
```

You will augment the run_experiment_ functions (representing scalar multiplication variations) to report the megaFLOPS achieved. You will need to use the high precision timer contained in the

support library. Given a square matrix with side length of `size` we can calculate FLOPS for scalar multiplication using the following equation (you will need to convert it to megaFLOPS - watch out for integer overflow):

$$FLOPS = (1 * size * size)/time$$

The support library provides a high precision timer of type `hptimer_t`, a function to get the current time get_time, and a function to take the difference of two `hptimer_t` time values in diff_timers. To get accurate timing you will need to run the operation many times (for this component at least 10,000 times), and take the average of all of those runs. You will likely want to add the averaging mechanism to your code, and report the average time, and average FLOPS. **You will also need a *quiet* machine** - make sure no other programs are running at the same time.

For each experiment, your code will look something like the following with the timer outside of the outermost loop:

```
1  double run_experiment_ij(mtype_t *matrix,
2                           mtype_t scalar, int N) {
3     int i, j, iter;
4     hptimer_t start, end;
5     double final_time;
6
7     start = get_time();
8     for( iter = 0; iter < MAX_ITERS; ++iter )
9        for(j = 0; j < N; ++j ) {
10          for(i = 0; i < N; ++i ) {
11             matrix[i][j] = scalar * matrix[i][j];
12          }
13       }
14    }
15    end = get_time();
16    final_time = diff_timers(start, end) / MAX_ITERS;
17    return final_time;
18 }
```

Generate results for NxN matrices where N ranges from 2 to 1024 by the powers of 2 (e.g., 2, 4, 8, 16, ...). Your program should take one optional command line parameter to specify the upper limit of the size of N (which may be larger than 1024). Generate a single graph containing the results for both experiments with the matrix size (in Bytes) on the x-axis, and MFLOPS on the y-axis. You will want to put the **x-axis in a log scale**.

Allocate the matrix once for each matrix size, and clear it before each call to the run_experiment_ function. To get the number of bytes in a single `mtype_t` variable use the `sizeof(mtype_t)` operation, just like you use when you dynamically allocate memory with malloc.

Questions

Review the hwloc data, and the graph generated from the experiments. The performance of the experiments should diverge at some matrix size. In your documentation answer the following questions:

- Which experiment produces the best overall performance?
- Describe the cause of the divergence of performance between the experiments.

Project 4.3 – Matrix Multiplication – (25 Points)

In this component of the assignment you will be implementing a matrix multiplication program in **six different ways**. This program will **generate three random matrices** and solve the equation $C = C + A * B$.

One variation of the program will implement the following code.

```
1  double run_experiment_ijk(mtype_t *matrix_a,
2                            mtype_t *matrix_b,
3                            mtype_t *matrix_c, int N) {
4     for( iter = 0; iter < MAX_ITERS; ++iter )
5        for(i = 0; i < N; ++i ) {
6           for(j = 0; j < N; ++j ) {
7              for(k = 0; k < N; ++k) {
8                matrix_c[i][j] += matrix_a[i][k] * matrix_b[k][j];
9              }
10          }
11       }
12    }
13 }
```

You will need to implement all possible permutations of the three loops above as different functions (giving you 6 different functions to write). You will use the same mechanisms to allocate, clear, and access array members so as with the scalar multiplication component. You will also be adding in the timing mechanism surrounding the outermost loop in each function, just as you did with the scalar multiplication component.

Given a square matrix with side length of `size` we can calculate FLOPS for matrix multiplication using the following equation (you will need to convert it to megaFLOPS - **watch out for integer overflow**):

$$FLOPS = (2 * size * size * size)/time$$

Generate results for NxN matrices where N ranges from 2 to 1024 by the powers of 2 (e.g., 2, 4, 8, 16, ...). Your program should take one optional command line parameter to specify the upper limit of the size of N (which may be larger than 1024). To get accurate timing you will need to run the operation many times (fewer than scalar multiplication, but at least 50 times), and take the average of all of those runs. You will likely want to add the averaging mechanism to your code, and report the average time, and average FLOPS. Generate a single graph containing the results for all experiments with the matrix size (in Bytes) on the x-axis, and MFLOPS on the y-axis. You will want to put the **x-axis in a log scale**.

Questions

Review the hwloc data, and the graph(s) generated from the experiments. The performance of the experiments should diverge at some matrix size. In your documentation answer the following questions:

- Which experiment(s) produces the best overall performance? Why?
- Rank the six experiments from best to worst in terms of MFLOPS.
- Describe the cause of the divergence of performance between the experiments.
- You may notice that some of the experiments *pair up*. Why does that happen?

Project 4 (Bonus) – Performance Boosters – (+5 Points Each)

Add a new experimental function to each of the scalar multiplication and matrix multiplication programs called run_performance_optimized. In this function you will be writing a new multiplication algorithm that has better performance than the best found during experimentation.

You have flexibility in the development of the new algorithm. You **may not** use any external mathematical libraries (other than the `math.h` library, if needed) to implement your algorithm. The implementation must use the same array and array access functionality used during the experiment (e.g., allocate_matrix, clear_matrix, GET_INDEX).

Your documentation must include a detailed description of how each of your algorithms work. Your discussion must also clearly explain **why** and **how** the algorithms improves the performance of the best prior versions. You will re-run the experiments to include the performance of the new algorithms. The graphs and data files should include these additional performance numbers.

Style & Documentation

Your assignment should be coded in a **consistent style** according to the **Style Requirements** handout.

Documentation is a critical piece of software development. Unfortunately, it is often left to the last minute and forgotten. In this assignment (and all of the assignments in this course) you will provide a single document containing at least the following pieces of information (each identified in the documentation as separate sections):

- Author(s), date
- Brief summary of the software
- How to build the software
- How to use the software
- How you tested your software.
- Examples
- Known bugs and problem areas
- Raw experimental data (**separate text file** - see next section for formatting details)
- Graphs of experiments
- Answers to questions posed as part of the assignment

Documentation should be in the form of a plain text or PDF document in the base of the assignment directory submitted. If you choose to create a PDF, it does not matter with what software you choose as long as it is easily readable. If you choose to create a plain text document then it must be named `README`, and is cleanly formatted so that it is easily readable.

You will need to include a the raw data from your experiments in **a separate text file** for each of the project components. You will also need to include a single graph representing that data for each of the project components (i.e., one graph for the scalar multiplication component and a separate graph for the matrix multiplication component). The graphs should be in either **PDF or PNG format**.

Testing

Testing is another important part of the software development life cycle. You must describe in your documentation how you tested your assignment to ensure it met the requirements of this project.

Useful Documentation & Hints

General advise on completing this assignment Start early, and ask questions. This assignment may seem fairly simple, but micro-benchmarking is a subtle art so be sure to review your code before submitting.

Note that some of the experiments may take quite a while to run (particularly for large values of N for matrix multiplication) – maybe an hour or so depending on your computer. Allow yourself sufficient time to both write the code and run the experiments.

Defensive programming is an important concept in operating systems: an OS cannot simply fail when it encounters an error, therefore it must check all input parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation. It is strongly recommended that you check the return code of all system and library function calls as this will often catch errors in how you are invoking them.

Test, test and test some more! The instructor will be extensively testing your assignment, so you should too.

Your program should report the data for each matrix side (`N`) as a single row of data. The first column is the matrix side (`N`), followed by the number of Bytes in the matrix, followed by a separate column reporting the average MFLOPS for each of the loop ordering experiments. Below is an example of the **expected formatting** for the scalar multiplication component.

```
shell$ ./scalarmult 8
2   32    234.775162055         228.271893438
4   128   266.224013401         255.202105041
8   512   270.716543185         288.774952226
```

You may find the following documentation useful in writing your program:

- From the **stdlib.h** library:
  - `strtol()`: Convert a string to an integer. (Do not use `atoi`).
  - `srandom(time(NULL))` : Seed the random number generator with the current time.
  - `random()` : Access a random, positive integer value.

Packaging & handing in your assignment

You will turn in a single compressed archive (either bzip/gzip'ed tar file, or zip file) of your completed assignment directory to the D2L Dropbox for each portion of the assignment. The compressed archive file should reference both your name, the assignment, and part of the assignment (e.g., **project4-jjhursey.tar.bz2**). The assignment directory should contain the following items:

- Documentation (either plain text or PDF)
- Raw data files
- Any additional graphs of the data
- A `Makefile` to build the software
- All of the necessary source files to compile your software