Lab 1a: Cilk++ warm-up

In this lab you will learn how to write parallel programs with Cilk++. In the first part of the lab, you will be introduced to the Cilk++ concurrency platform and learn how to use the Cilkview and Cilkscreen tools.

You should spend 4 hours or less on this lab, and you should try to do most of the work minus write-up by class on Thursday. Since Labs 1b and 1c will be handed out on this Thursday and next Tuesday, respectively, and will require more of your time, do not spend too much time on Lab 1a.

Although you should do all the steps of this lab, you need not submit documentation demonstrating that you did every step. Since the focus of the class is on independent discovery, you should feel free to investigate what you see as the interesting issues and document those. You should not wander too far from the topic of the lab, however. For example, for this part of the lab, it suffices to provide a few paragraphs describing what you learned while devising your best matrix-multiplication code and hand in a Cilkview plot documenting the performance of your code.

We will also upload this lab handout to NB. As one requirement for this lab, we would like you to make at least one comment about some aspect of this lab by using NB (`http://nb.csail.mit.edu`) to annotate this lab handout (`lab1a.pdf`).

If you don't have a basic familiarity with C/C++, here are some on-line tutorials which might prove useful:

- `http://www.cplusplus.com/doc/tutorial/`
- `http://www.learncpp.com/`
- `http://www.cprogramming.com/tutorial.html#c++tutorial`
- `http://www.cs.wustl.edu/~schmidt/C++/`

A tutorial on C++ will also be scheduled next week for those interested.

Getting Started

Before beginning this lab, please make sure you have followed the instructions on the course website for setting up and configuring access to the class resources, outlined in Handout 3, "Getting Started."

We have given you a git repository with the code for this portion of the lab. You can obtain this code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab1a/ lab1a
```

Once this repository has been cloned, you should see two subdirectories within the new directory `lab1a`: `qsort` and `mm`.

We are giving you a git repository for the lab to give you easy access to version control tools as you experiment with the lab code. There is no need to submit code for this part of the project via git; the repository is solely for your personal use. For more information on using git, please take a look at the Git user's manual at `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`.

Helpful Hints

- When running a Cilk++ program, you may specify the number of worker threads to `N` using the `-cilk_set_worker_count=N` command line parameter (defaults to the number of cores on the system).
- When using Cilkview or Cilkscreen, you should run your programs on smaller inputs than when simply running them standalone since there is a large performance penalty for instrumentation.
- You can use the `-help` flag with Cilkview or Cilkscreen to see an explanation of all command line flags for these programs.

1 Examining a Parallel Quicksort

In this problem, you will experiment with an existing Cilk++ project, and learn how to use the Cilkview Performance Analyzer and Cilkscreen Race Detector. You should make running your Cilk applications through Cilkview and Cilkscreen a standard practice.

1.1 Measuring performance with Cilkview

The sources for this exercise can be found in the `qsort` subdirectory. Build the qsort binary by running make. This will produce a parallel quicksort binary. The binary takes two optional arguments. The first specifies the number of data points (defaults to 10 million), and the second specifies the number of trials to run (defaults to 1).

(a) Run quicksort with the 10,000,000 data points (the default) using 1 through 12 threads using the Cilkview tool:

```
cilkview -trials all 12 ./qsort.64
```

What are the execution times? What happens when you run qsort with more threads than there are processors on the cagnode machines (8)?

Cilkview Tip: Other active processes on the cagnodes may affect the observed performance of your program in unexpected ways. To mitigate this effect, you may wish to run trials on your program multiple times and record the best performance observed for some trial over all runs. Cilkview can record the best performance seen for some trial automatically using the `-append` flag as follows:

```
cilkview -trials all 12 -append ./qsort.64
```

For more information on this feature, see the Cilk++ Programmer's Guide.

Cilkview generates a speedup graph with parallelism bounds and measured speedup. If you enable X11 forwarding when logging into the cagnodes, you will see this graph automatically displayed when Cilkview terminates. (If you do not you may see some harmless warning concerning wxWidgets and "difficulty fitting plot titles into key.")

The results of your tests are saved to a file, so you can load the results into `gnuplot` and manually view the speedup graph later. To do this, login to one of the cagnodes with X11 forwarding enabled, run `gnuplot`, and enter `load "qsort.plt"`. (You must place quotes around

`qsort.plt` in order for gnuplot to parse the command correctly.) Alternatively you may copy the files `qsort.csv` and `qsort.plt` to your Athena home directory, log in to an Athena cluster machine, run `gnuplot`, and enter `load "qsort.plt"`.

Cilkview Tip: You may wish to generate an eps file of your Cilkview speedup graph for inclusion in your write-up. You can make `gnuplot` generate this file for you by adding the following lines to the top of `qsort.plt`:

```
set terminal postscript eps color
set output "qsort.eps"
```

Loading this modified `qsort.plt` file into `gnuplot` will cause `gnuplot` to create the file `gsort.eps`, which you may then include in your write-up.

1.2 Detecting races with Cilkscreen

(b) Uncomment the following line near the top of the file to introduce a race condition in the parallel code:

```
#define INTENTIONAL_RACE
```

Look at the code enabled by this change and explain how the race could cause quicksort to fail to sort the array of integers.

(c) Run qsort through Cilkscreen using the following command:

```
cilkscreen ./qsort.64 1000
```

Is Cilkscreen able to detect the race? How many races does Cilkscreen detect? How many races does Cilkscreen detect when sorting 10000 integers?

Note that Cilkscreen runs the program using a single worker, and therefore will experience no multicore speedup. For more information on Cilkscreen and its effect on performance, see the Cilk++'s Programmer's Guide.

2 Matrix Multiplication

In this part, you will write a multithreaded program in Cilk++ to implement matrix multiplication. One of the goals of this assignment is for you to get a feeling of how ***work***, ***span***, and ***parallelism*** affect performance. First, you will parallelize a program that performs matrix multiplication using three nested loops. Then, you will write a serial program to perform matrix multiplication by divide-and-conquer and parallelize it by inserting Cilk keywords. Finally, you will implement a parallel version of Strassen's algorithm.

For those of you who have not looked at matrix multiplication in a little while, the problem is to compute the matrix product

$$C = AB \ ,$$

where C, A, and B are $n \times n$ matrices. Each element c_{ij} of the product C can be computed by multiplying each element a_{ik} of row i in A by the corresponding element b_{kj} in column j in B, and then summing the results, that is,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ .$$

For more information on matrix multiplication, please see `http://en.wikipedia.org/wiki/Matrix_multiplication#Ordinary_matrix_product`.

The nested-loop and divide-and-conquer versions of these programs can be adapted to work with arbitrary rectangular matrices. To simplify the interface, however, we limit ourselves to $n \times n$ square matrices where n is an exact power of 2.

2.1 Matrix multiplication using loop parallelism

The sources for this exercise can be found in the `mm` subdirectory.

The file `mm_loops.cilk` contains two copies of a $\Theta(n^3)$-work matrix multiplication algorithm using a triply nested loop. The first copy (`mm_loop_serial`) is the control for verifying your results — leave it unchanged. The second copy (`mm_loop_parallel`) is the one that you will parallelize. This file also contains a test program that verifies the results of your parallel implementation and also provides infrastructure for timing and measuring parallelism.

(d) Compile `mm_loops` using make, and verify that it operates correctly. Supply the `--verify` command-line option to force running all tests.

(e) Now parallelize the `mm_loop_parallel` function by changing the outermost `for` loop into a `cilk_for` loop. Verify correct results with the `--verify` option. Run Cilkview on your program and report the theoretical and actual speedup. Do not use the `--verify` option when running Cilkview.

(f) Change the outermost `cilk_for` back into a serial `for` loop and change the middle `for` loop into a `cilk_for` loop. Repeat the test with the `--verify` option, and then report the results from Cilkview. Did any results change? Try making both loops parallel. Which of these combinations produces the best results?

2.2 Matrix multiplication by divide-and-conquer

Divide-and-conquer algorithms often run faster than looping algorithms, because they exploit the microprocessor cache hierarchy more effectively. This section asks you to write a divide-and-conquer implementation of matrix multiplication. You will find the source code for the incomplete program in `mm_recursive.cilk`. The program contains two implementations of matrix multiplication. The `mm_loop_serial` function is the same as before and is provided for verification and timing comparisons. The `mm_recursive_parallel` function is the skeleton of a divide-and-conquer implementation.

Your recursive implementation will be based on the identity

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix},$$

where A_{11}, A_{12}, etc., are submatrices of A. In other words, matrix multiplication can be performed by subdividing each matrix into four parts, then treating each part as a single element and (recursively) performing matrix multiplication on these partitioned matrices. (The number of columns in A_{11} must match the number of rows in B_{11}, and so forth.) Although the algorithm operates recursively, its work is still $\Theta(n^3)$, the same as the straightforward algorithm that employs triply nested loops.

(g) Compile `mm_recursive`, and verify that it compiles but fails to run successfully when using the `--verify` command-line argument. The failure is caused by the fact that the `mm_recursive_parallel` has not been fully implemented yet.

(h) In the file `mm_recursive.cilk`, fill in code in the `mm_internal` function to implement the divide-and-conquer algorithm. The error-prone task of subdividing the matrices into four parts has been done for you. All you need to do is to fill in the recursive calls (eight in total – one for each of the eight matrix-multiplications in the algorithm). Make sure to carefully read the comments in `mm_internal` before you begin. Compile and run your new `mm_recursive` program and verify that it runs successfully.

(i) Which recursive calls to `mm_internal` may be legally executed in parallel with one another and why?

Make your recursive function parallel by adding the `cilk_spawn` keyword in front of some of the recursive calls. You will need to add calls to `cilk_sync` as well in order to separate recursive calls that would otherwise cause a data race and to ensure that all of the work is complete before returning from the function.

Compile and run your new `mm_recursive` program. Verify that it is correct and report the results given by Cilkview. For large matrices, how does the performance of the recursive algorithm compare with the nested-loops algorithm on a single processor?

(j) Uncomment the line near the top of the program that reads:

```
#define USE_LOOPS_FOR_SMALL_MATRICES.
```

This change causes the algorithm to change from divide-and-conquer recursion to a triply nested loop for small matrices. How does this change impact performance? How does the performance of this new version compare to the previous versions?

2.3 Matrix multiplication by Strassen's algorithm

For large matrices, Strassen's algorithm can outperform the other two methods we've seen, because it entails $\Theta(n^{\lg 7}) = O(n^{2.81})$-work, rather than $\Theta(n^3)$. Section 2.3.1 describes Strassen's algorithm in detail (although it doesn't provide insight as to how Volker Strassen discovered this bizarre method). Section 2.3.2 describes the exercise itself.

2.3.1 Description of Strassen's Algorithm

Strassen's algorithm can multiply two $n \times n$ matrices in $\Theta(n^{\lg 7}) = O(n^{2.81})$ work, which is asymptotically better than more straightforward methods that require $\Theta(n^3)$ work. Let A and B be two $n \times n$ matrices. For the rest of the assignment, assume that n is an exact power of 2. Recall that the product of A and B is defined to be $C = AB$, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \ .$$

Although this definition leads to a straightforward $\Theta(n^3)$-work algorithm to compute the product, a remarkable identity can be exploited which leads to a divide-and-conquer algorithm with work $\Theta(n^{\lg 7}) = O(n^{2.81})$.

Partition C, A, and B as follows:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, \quad A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Then, we have

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Instead of doing the usual divide-and-conquer, perform the following calculations:

$$\begin{aligned} M_1 &= (A_{11}+A_{22})(B_{11}+B_{22}), \\ M_2 &= (A_{21}+A_{22})B_{11}, \\ M_3 &= A_{11}(B_{12}-B_{22}), \\ M_4 &= A_{22}(B_{21}-B_{11}), \\ M_5 &= (A_{11}+A_{12})B_{22}, \\ M_6 &= (A_{21}-A_{11})(B_{11}+B_{12}), \\ M_7 &= (A_{12}-A_{22})(B_{21}+B_{22}). \end{aligned}$$

We can now express C in terms of the M's as follows:

$$\begin{aligned} C_{11} &= M_1+M_4-M_5+M_7, \\ C_{12} &= M_3+M_5, \\ C_{21} &= M_2+M_4, \\ C_{22} &= M_1-M_2+M_3+M_6. \end{aligned}$$

If all intermediate results are appropriately saved, a multiplication of size n can be reduced to 7 multiplications of size $n/2$, plus 18 matrix additions.

2.3.2 Matrix-multiplication using Strassen's algorithm

(k) Modify your divide-and-conquer solution implement Strassen's algorithm. Verify its correctness, run cilkscreen and run cilkview.

For some applications, Strassen's algorithm produces lower-quality answers than the standard algorithm does due to loss of precision from round-off error. You may not actually see round-off error using our test code, since the matrices are initialized with random integers.