Parallel Algorithms (15-492), Fall 07
Assignment #5 **Due: November 13th**

1 Background

In the first half of the course, we saw a number of problems that can be solved efficiently in parallel. Even though these solutions seem simple, it takes another step of thinking to realize them as an efficient implementation. In this assignment, you will get to experience this process first hand.

This is a 2-week long programming assignment. There are 2 problems on this assignment. It will be due at the beginning of class on Tuesday November 13, 2007.

For each problem, you will provide solutions in 4 different programming environments: Cilk, X10, OpenMP, and POSIX threads (pthreads). In preparation for the final project, these exercises aim at acclimating you to different parallel-programming paradigms. As will be apparent, each paradigm has its own benefits and drawbacks that you should carefully take into account when working on the final project.

To reward good design and implementation, your grade will be determined in part by the performance of your code. Although there is no hard threshold to receive a full credit, turning in embarrassingly sequential code almost certainly warrants low score.

Hand-in Directory. To hand in your solutions, copy all the files into

`/afs/cs.cmu.edu/academic/class/15492-f07/handin/`*`userid`*

where *`userid`* is your Andrew ID.

Multi-core Machine. You will soon receive an e-mail from us with your login name and password for use on multi6, a generous donation from Intel Corp. The machine (multi6.aladdin.cs.cmu.edu) is a Linux box with 4 dual-core 3.4Ghz Intel Xeon processors and 32 GB of RAM. This should be plenty of resource for you to experiment with parallel programming and to use for your project.

The best way to access this machine is via ssh. Your home directory will be `/usr0/15492-f07/`*`userid`*. While you are at it, please feel free to look around. We will support the following platforms: X10, Cilk, NESL, pthreads, and OpenMP.

If you have access to a multi-core machine, it might be fun to try out these packages on your machine; however, we probably will not be able to assist you if a problem arises. If you work on this assignment on a local machine, keep in mind that when grading your assignment, we will run your code on multi6.

2 Matrix Multiply

In this problem, you will provide an efficient implementation of the matrix-multiply routine, optimized for a multi-processor, shared-memory machine. Your program will meet the following specifications:

- **Program Names:** You will hand in the following files:

  `mat_mult.x10`, `mat_mult.cilk`, `mat_mult_omp.{c,cxx,cc}`, `mat_mult_pthr.{c,cxx,cc}`.

- **Command-line Arguments:** We will run your program with the following arguments "`-p <p> <file.mat> <output.mat>`" where `<p>` is the number of processors available , `<file.mat>` is the

input file, and `<output.mat>` is the output file. We supply the number of processors in case your algorithm will take this information into account.

- **Input:** The program will read from a file (supplied through a command-line argument) a number n, and two matrices A and B, both of dimensions $n \times n$. We will assume that n is a power of two. A sample input file is provided below. The first number is n. The n subsequent lines are entries of A, and the next n lines are entries of B. It is safe to assume that these entries are floating numbers (think C's double) with absolute values less than 10^3.

sample.mat	output.mat
4	4
1 3 4.2 5	64.4 85.8 33.3 57.6
2 4 5 6	84.0 112 41.4 72.8
5 8 1 5	123 140 31.8 61
2 2 1 2	42 50 13.2 26.8
8 8 1 4.4	
7 7 2.1 1	
2 4 5 6	
5 8 1 5	

- **Output:** As suggested in the example above, you will print out the matrix $A \times B$. The first line is the dimension of $A \times B$. The following n lines are entries of $A \times B$. We will not use `diff` to grade your answer. Our check script will tolerate floating-point errors up to 10^{-5}.

3 Scan

Back in NESL, `scan` was a primitive we could invoke happily at ease. In this problem, we will build the `plus_scan` primitive from scratch. Here are some details.

- **Program Names:** You will hand in the following files:

  `plus_scan.x10`, `plus_scan.cilk`, `plus_scan_omp.{c,cxx,cc}`, `plus_scan_pthr.{c,cxx,cc}`.

- **Command-line Arguments:** We will run your program with the following arguments "`-p <p> <array.dat> <output.dat>`" where `<p>` is the number of processors available, `<array.dat>` is the input file, and `<output.dat>` is the output file. Again, we tell you the number of processors just in case your algorithm wants to take this information into account.

- **Input:** The program will read from a file (supplied through a command-line argument) a number n, and n floating-point numbers. As shown in a sample input file below, the first number is n; the n numbers (one per line) are $x_1, \ldots, x_n$. It is safe to assume that these entries are floating numbers (think C's double) with absolute values less than 10^6.

array.dat	output.dat
6	6
4	0.0
2.2	4.0
3.1	6.2
5	9.3
7	14.3
1	21.3

- **Output:** As suggested in the example above, you will print the array's length n and the result of plus scan (one number per line). Again, our check script will tolerate floating errors up to 10^{-10}.