Prof. Charles E. Leiserson
Prof. Michael Bender
Prof. Hsu Wen Jing
Dr. Bradley C. Kuszmaul

Massachusetts Institute of Technology
6.895: Theory of Parallel Systems
Handout 2
September 8, 2003

Lab 1 — Cilk Programming

This assignment is due on *Monday, September 29.* In this assignment, you will write a high-performance multithreaded program to implement matrix multiplication. In particular, you will write a Cilk program to implement Strassen's algorithm. First, you will write a fast serial matrix-multiplication program to use as the base case in the recursion. Then you will parallelize your program using Cilk. One of the goal of this assignment is for you to get a feeling of the tradeoffs among work, space, and critical path. The fastest programs will win prizes. This lab can be done in groups of up to 3.

1 Serial matrix multiplication

This section asks you to write the fastest serial implementation of matrix multiplication that you can. You can find the source code for a simple matrix-multiplication program, called `mm_simple.c`, on the course web page. This program multiples two $n \times n$ matrices $O(n^3)$ time. The web page also provides a little test program called `mm_test.c`. The source code for `mm_simple.c` is reproduced, for your convenience, in Figure 1.

(a) Analyze and measure the performance of `mm_simple`. First, compile `mm_simple`, and verify that it works.

```
gcc -O2 -g mm_test.c mm_simple.c -o mm_simple && ./mm_simple
```

Now, modify the test code to measure the performance on some bigger matrices, and measure the performance. How fast is the program as the matrix size grows? A simple answer would be something like, "$10n^3$ microseconds, where n is the matrix size." Is the answer that simple?

Perform these studies on `ygg.lcs.mit.edu`, a 32-processor SGI Origin. Or, you may perform these studies on other machines. Any recently built Pentium machine is likely much faster than one processor of the Origin.

(b) Speed up the matrix-multiplication code, but do not change the data structure (nor should you try to make it multithreaded.) How fast can you get it to run?

Hint: Here are some ideas that may or may not work for speeding up the code:

1. The code in Figure 1 performs $O(n^3)$ stores into matrix C. Try rearranging the code so that each element of C is written only once.
2. Try taking further advantage of the processor's registers, e.g, by open-coding small 2×2 matrix multiplications in the inner loop.

```
#include "mm_simple.h"
/* Copyright(C) 2003 Bradley C. Kuszmaul.
 * This code is licensed under the GPL. */

/*********************************************
 * Effect: perform matrix multiply C = A*B.
 *   a is a pointer to the first element of matrix A, an m by n matrix.
 *   b is a pointer to the first element of matrix B, an n by k matrix.
 *   c is a pointer to the first element of matrix C, an m by k matrix.
 *
 *   The matrices are stored in column-major order.  That is A[I,J] is
 *   adjacent to A[I+1,J] in memory.
 *
 *   Rationale: Why lay out the arrays in column-major order, the way
 *   FORTRAN does, instead of in row-major order the way C would if you
 *   wrote
 *     double a[N][M];   ?
 *   Answer: Mainly because we want this code to be able to interoperate
 *   with FORTRAN.
 *
 */
void mm_simple (int m, int n, int k, double *a, double *b, double *c)

{
#define A(I,J) a[(I)+(J)*n]
#define B(J,L) b[(J)+(L)*k]
#define C(I,L) c[(I)+(L)*k]
    int i,j,l;
    for (i=0; i<n; i++) {
        for (l=0; l<k; l++) {
            C(i,l) = 0.0;
        }
    }
    for (i=0; i<n; i++) {
        for (l=0; l<k; l++) {
            for (j=0; j<m; j++) {
                C(i,l) += A(i,j) * B(j,l);
            }
        }
    }
}
```

Figure 1: A simple matrix-multiplication program `mm_simple.c`.

3. Try to exploit the cache by "blocking" your code. That is, rearranging the computation to use the cache more effiiently. On the MIPS, a cache line is 128 bytes, which is 16 double-precision floating-point numbers.
4. Use the `-S` option to `gcc` and look at the assembly language produced by the compiler. See whether rearranging the source code produces a faster exectuable. Remember that memory references are expensive and register operations are cheap.
5. Testing is important. We suggest that you incorporate the original simple and slow matrix-multiplication programs into your test code. Then, write a test program that automatically verifies that your modified matrix-multiplication code produces the same answer as the original code. One good idea for testing is to use large random matrices as inputs.

(c) Reorganize the data into a cache-oblivious data structure. How fast can you get it to run?

(d) *For the adventurous:* On some machines, such as a Pentium, you can get big speedups by using the vector operations provided in assembly language. On the Pentium there are the MMX and SSE instructions, for example. For an extra adventure, use SSE with a cache-oblivious data structure.

Hand in the best serial program you obtain, together with a plot of your experiments graphing running time against matrix size. Graph your performance formula on the same plot.

You may find that a tool such as Gnuplot, MATLAB, Octave, or Excel is useful for creating plots. Most such tools include a curve-fitting utility, which can help construct performance models.

2 Cilk warm up

This section asks you to some short and easy Cilk programs so that you get accustomed to Cilk programming. You need not do this section, but doing it may actually speed your progress no later sections.

(e) Implement the divide-and-conquer matrix-multiplication algorithm described in class. Recall that the algorithm presented in class is based on the identity

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

Make sure your algorithm has $O(\lg^2 n)$ critical path. (You will need to parallelize the additions.)

(f) Implement a matrix-multiplication algorithm with $\Theta(n^3)$ work that does not use extra space.

(g) Implement a matrix-multiplication algorithm with $\Theta(n^3)$ work and $\Theta(\lg n)$ critical path, and compare the performance of the three programs.

3 Strassen's algorithm

In this section we present Strassen's algorithm for multiplying two $n \times n$ matrices.[1] The remarkable property of the algorithm is that it is asymptotically faster than the naïve $O(n^3)$ algorithm.

Let A and B be two $n \times n$ matrices. For the rest of the assignment, assume that n is a 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 $O(n^3)$ algorithm to compute the product, a remarkable identity can be exploited which leads to a faster divide-and-conquer algorithm. Partition A and B as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad \text{and} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} .$$

Then we have:

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

where

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \ , \\ C_{12} &= w + (A_{21} + A_{22}) \cdot (B_{12} - B_{11}) + (A_{11} + A_{12} - A_{21} - A_{22}) \cdot B_{22} \ , \\ C_{21} &= w + (A_{11} - A_{21}) \cdot (B_{22} - B_{12}) - A_{22} \cdot (B_{11} - B_{21} - B_{12} + B_{22}) \ , \\ C_{22} &= w + (A_{11} - A_{21}) \cdot (B_{22} - B_{12}) + (A_{21} + A_{22}) \cdot (B_{12} - B_{11}) \ , \\ w &= A_{11} \cdot B_{11} - (A_{11} - A_{21} - A_{22}) \cdot (B_{11} - B_{12} + B_{22}) \ . \end{aligned}$$

If all intermediate results are appropriately saved, a multiplication of size n can be reduced to 7 multiplications of size $n/2$, plus 15 matrix additions. The relevant steps are outlined below (the S_i's, M_i's and R_i's are temporary matrices):

$$\begin{aligned} S_1 &\leftarrow A_{21} + A_{22} \\ S_2 &\leftarrow S_1 - A_{11} \\ S_3 &\leftarrow A_{11} - A_{21} \\ S_4 &\leftarrow A_{12} - S_2 \end{aligned}$$

[1] This particular version of the algorithm is actually due to S. Winograd.

$$
\begin{aligned}
S_5 &\leftarrow B_{12} - B_{11} \\
S_6 &\leftarrow B_{22} - S_5 \\
S_7 &\leftarrow B_{22} - B_{12} \\
S_8 &\leftarrow S_6 - B_{21} \\
M_1 &\leftarrow S_2 \cdot S_6 \\
M_2 &\leftarrow A_{11} \cdot B_{11} \\
M_3 &\leftarrow A_{12} \cdot B_{21} \\
M_4 &\leftarrow S_3 \cdot S_7 \\
M_5 &\leftarrow S_1 \cdot S_5 \\
M_6 &\leftarrow S_4 \cdot B_{22} \\
M_7 &\leftarrow A_{22} \cdot S_8 \\
R_1 &\leftarrow M_1 + M_2 \\
R_2 &\leftarrow R_1 + M_4 \\
C_{11} &\leftarrow M_2 + M_3 \\
C_{12} &\leftarrow R_1 + M_5 + M_6 \\
C_{21} &\leftarrow R_2 - M_7 \\
C_{22} &\leftarrow R_2 + M_5
\end{aligned}
$$

The 7 intermediate multiplications can be computed recursively by the same algorithm. The work obeys the recurrence

$$T_1(n) = 7T_1(n/2) + \Theta(n^2) \ ,$$

whose solution is $T_1(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$. More details on Strassen's algorithm can be found in [1, Chapter 31].

(h) Describe an implementation of Strassen's algorithm with a short asymptotic critical-path length. In other words: identify which parts can be done in parallel, and analyze the critical path of the resulting parallel algorithm.

(i) Analyze the space required by the implementation you described in part (h).

4 Implementation of Strassen's algorithm

You are now ready for the main part of the assignment.

(j) Implement a simple Cilk version of the algorithm from part (h). This program should really be simple, just to get you going. Do all the recursive multiplications in parallel, and try to do some of the additions in parallel, too.

(k) In this part, we ask you to optimize your program. Experiment with *at least* one of the following ideas.

1. Experiment with tradeoffs between parallelism and space. The idea is that if you force certain operations to happen after other operations (with a `sync`), you may be able to reuse memory, at the expense of a longer critical path. Reusing memory may increase your speed, because of better cache behavior.

   If your changes alter the (asymptotic) critical path and space requirements of your program, analyze them again.

   You can use the Cilk `SYNCHED` predicate to find out at runtime whether previous computations have already terminated. In this case, you may be able reuse some space.
2. Instead of recursing down to $n = 1$, switch to the $O(n^3)$ algorithm for small n. Find the best switch point. Determine experimentally the constants involved in the $O(n^3)$ expression and in your expression for the total work, and analyze what the optimal switch point should be. Do your predictions agree with the experiments?
3. Present-day computers are faster when they can access memory sequentially. The matrix product AB is bad from this point of view, since if both A and B are stored in row-major order, then the inner loop will need to access the columns of B, which are not stored sequentially in memory. Modify your program to compute AB^T instead.
4. Alternatively, suggest other improvements and experiment with them.

Hand in your simple Cilk implementation of Strassen's algorithm, as well as your optimized program, together with a performance plot and a performance model of both programs. Be sure to include sufficient documentation to explain what is interesting about your program.

Fast and/or original programs will win a prize! Prize categories will include "fastest serial program written in portable C", "fastest portable Cilk program", and "fastest on any machine in any language", "fastest on ygg", "fastest on a PC", "fastest cache-obvlivous serial program" and others.

A Running Cilk on `yggdrasil.lcs.mit.edu` (ygg)

We set up a directory `/xlv0/people/6.895/lab-1` on `ygg`, containing files you may find useful. Moreover, Cilk is installed on `ygg` in the directory `/usr/freeware2/bin/cilkc`, and there is a directory of examples you can look at (`/xlv0/people/6.895/cilk/`). These examples are described in the Cilk manual.

Cilk needs a version of `gcc` *with the GNU linker to run.* The right version of `gcc` is installed in `/usr/freeware2/bin`. Please set up your `PATH` as described in the file `dot.login` in the lab directory. If you run on a Linux machine, you don't need to worry about this.

The lab directory contains a file `strassen.cilk`, that provides a testing framework for the program you write. Also, you should use the `Makefile` we provide.

The test program takes a couple of options. The option `-n <n>` determines the size of the matrix to be tested. By default, the matrix is 256×256. The test program creates a random matrix and runs your multiplication routine once, telling you the run time, the work and the critical path. The other option is `-c`, which compares the output of your multiplication routine with the correct result, and tells you whether they are the same.

In addition, every Cilk program accepts a predefined set of Cilk options. These are explained in the Cilk manual. You should at least be aware of the options `-nproc <n>`, that sets the number of processors your program is running on. For example:

```
% ./strassen -nproc 8 -n 1024
```

Remember that Cilk options must precede program-specific options in the command line.

B Linux Kernel Patch

If you have root access on a Linux machine, you can obtain and install the estime kernel patch from `http://estime.sourceforge.net/`, which provides high-precision low-overhead virtual timers. These timers can help you understand exactly how much runtime your program consumes. To install estime on your machine you must recompile the kernel. To use estime on your serial program, you must add calls to the timers from your program.

Contact Bradley `bradley@mit.edu` if you want to try using an estime-enabled version of Cilk on your estime-enabled kernel.

C Conventions used in the test program

You have to put your multiplication routine into the `strassen.cilk` program. As provided, the program calls a sequential routine. Your goal is to write the function

```
cilk void strassen(int n, REAL *A, int an, REAL *B, int bn,
                   REAL *C, int cn)
```

Your function must compute $C = AB$, where A, B and C are $n \times n$ real matrices, and n is a power of two. The type `REAL` is predefined to `double`. The matrices are stored in the following way. Every matrix (say, `A`) has an associated integer (`an`). `A` points to the first element of the matrix, and `an` is the distance between two elements in the same column. In other words, the element A_{ij} is stored in location `A[i * an + j]`. The macro `ELEM(A, an, i, j)` is predefined to be `A[i * an + j]`, for convenience.

In the test program there are a bunch of utility functions that allocate and destroy matrices, print, compare two matrices, etc. Their use and implementation are straightforward, and documented in the code. We provided a sequential function `matrixmul`, that computes the product of two matrices. That function is likely to be slow, but it is useful for checking the result of your program.

D Accurate measurements

The machine `ygg` is a 32-processor machine shared by everyone. If other people are running it can be difficult to get good timings on your program. you can use the `w` command to find out who is using the system.

```
ygg% w
 12:58pm  up 45 days, 23:32,  2 users,  load average: 0.00, 0.00, 0.00
User     tty from            login@   idle   JCPU   PCPU  what
bradley  q0  bradley.lcs.mi 12:52pm                      w
seanlie  q3  cagfarm-20.lcs Sat10am  23:10     18         -tcsh
```

If the load average is close to 0, you can safely time your program.

If you perform some of your initial development on another machine, be aware that different things are fast on different machines. For example, the MIPS processors used on `ygg` have 32 registers for floating point computations, but a Pentium has only a few registers. Thus, you may find two programs X, and Y, for which X is faster than Y on `ygg`, but Y is faster than X on a pentium.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw Hill, 2nd edition, 2001.