Massachusetts Institute of Technology
6.892: Theory of Parallel Systems
Professor Charles E. Leiserson

Handout 17
March 6, 1997

# Lab 2 — Cilk programming

This assignment is due in class on *Thursday, March 20.* The goal of the problem is threefold. First, you will learn Cilk. Second, we want you to get a feeling of the tradeoffs among work, space and critical path. Finally, the fastest program will win a Cilk T-shirt.

In this assignment you will implement Strassen's algorithm for matrix multiplication in Cilk.

## 1 Warm up

In this section you will write some short and easy Cilk programs, so that you get accustomed with Cilk.

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

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

Make sure your algorithm has $\Theta(\lg^2 n)$ critical path; you need to parallelize the additions in order to achieve that.

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

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

## 2 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

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

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

The definition leads to a straightforward $O(n^3)$ algorithm to compute the product. However, there is a remarkable identity that 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

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

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):

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

$$
\begin{aligned}
C_{11} &\leftarrow M_2 + M_3 \\
C_{12} &\leftarrow R_1 + M_5 + M_6 \\
C_{21} &\leftarrow R_2 - M_7 \\
C_{22} &\leftarrow R_2 + M_5
\end{aligned}
$$

Clearly, 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 Chapter 31 of CLR.

**(d)** 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.

**(e)** Analyze the space required by the implementation you described in part (a).

# 3 Implementation of Strassen's algorithm

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

**(f)** Implement a simple Cilk version of the algorithm from part (a). This 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.

**(g)** 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.

*Fast and/or original programs will win a prize!*

# A Running Cilk on `xolas`

We set up a directory `/usr/local/6.892/lab-2` on `xolas`, containing files you may find useful. Moreover, Cilk is installed on `xolas` in the directory `/usr/local/6.892/cilk`, and there is a directory of examples you can look at. 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/local/cilk/bin`*. Please set up your* `PATH` *as described in the file* `dot.login` *in the lab directory.*

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 $256 \times 256$. 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`, that compares the output of your multiplication routine with the correct result, and tells you if they are the same or not.

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, and `-heap <n>`, that determines how much memory your program is allowed to use (in megabytes). The default is 1 processor and 64 megabytes. For example:

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

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

# B 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. We also provided a faster function `matrixmul2`, that works only for even $n$. You may use either in your program, if you wish.

# C `xolas` usage policy

When you log onto `xolas`, the message of the day tells you which machines are available for use. In general, you should only use `xolas0` and `xolas1` for compiling/debugging.

In order to get reliable times, you should run on an idle machine. The recommended way is to use an experimental queuing system that we recently set up. You can submit jobs using the command `/usr/local/bin/submit`:

```
submit </full/pathname/of/program arguments>
```

Remember to tell submit the full path name of your program. This can usually be done with something like

```
submit $PWD/fib -nproc 4 30
```

You can view which of your jobs are in the queue using `submit -l`. You can remove jobs from the queue using

```
submit -r <job id returned from 'submit -l'>
```

The standard output of your program will be mailed to you.

*Please, do not subbit more than two jobs at once.*

The queuing system is still experimental. Complaints should be directed to `fdonovan@xolas`. Extra credit will be awarded to people who are able to acquire Unix superuser status using `submit`.

The machine `xolas7` is unique in that it has 12 processors and 1GB of memory. You can also use `xolas7` for measuring execution times. If somebody else is using the machine, the measurements will be inaccurate. You can get an idea of who is using a machine by typing `w`:

```
xolas7% w
 10:26am  up 16:46,  2 users,  load average: 0.00, 0.00, 0.01
User     tty           login@  idle   JCPU   PCPU  what
athena   pts/0        10:26am                      w
```

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