Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.827 Multithreaded Parallelism: Languages and Compilers

Problem Set 1

Out: 21 September 1998
Due: 30 September 1998

This problem set is intended to give you a taste of programming in Cilk, an explicitly parallel multithreaded language. The problems involve writing and parallelizing several C programs. If any of you are unfamiliar with C, please talk to the TA's.

First, take the time to skim through the Cilk manual and familiarize yourself with the Cilk system. For the moment, you are going to be running your Cilk programs as multiple processes on single-CPU machines (for the purposes of the problem, this method should give you more than enough data).

Please Remember: 1) You can work in groups of up to three people; include the names of all group members on all problems. 2) Turn in the answer to each problem as a separate packet. 3) Comment your code carefully and include output from sample runs.

At the moment only the Sun version of Cilk is installed on Athena. In order to use it, do the following:

```
add gnu
add 6.827
```

If you want to play around with settings, poke around in the course locker. The full Cilk distribution can be found there. Note that it would be non-trivial to get the distribution running on non-Sun architectures (more due to the structure of the system than any technical difficulty). Note that `cilk2c` is a shell script which sets appropriate environment variables. There is also a script called `cilkmake` which is designed to build Cilk programs given a list of source file names.

Problem 1 — Numerical Integration

In this first problem, we examine two simple algorithms for numerical integration based on Simpson's rule.

Part a:

Given a function f and an interval $[a, b]$, Simpson's rule says that the integral can be approximated as follows:

$$I = \frac{h}{3}\left[f(a) + 4f(a+h) + f(a+2h)\right]$$

where

$$h = \frac{b-a}{2}$$

For better accuracy, the interval of integration $[a, b]$ is divided into several smaller subintervals. Simpson's rule is applied to each of the subintervals and the results are added to give the total integral. If the subintervals of $[a, b]$ are all of the same size p, where $p = 2h$, then we have the *Composite Strategy* for integration.

Write a C program containing a function

```
double composite_strategy(double (*f)(double),double a,double b,int n);
```

where

f is the function to integrate

a,b are the endpoints of the integration interval

n is the number of subintervals such that $h = (b - a)/2n$

Integrate some simple functions and try different values for n.

Part b:

Depending on the value of n, there can be a lot of parallelism in the implementation of the *Composite Strategy.* Write a version of the `composite_rule` in Cilk that parallelizes the subinterval computations. Be careful though: too many parallel activities can lead to substantial overhead!

Part c:

If the subintervals of the integration are not all equal and can be changed as required, then we obtain an *Adaptive Strategy* for integration. A simple adaptive integration algorithm can be described as follows:

1. Approximate the integral using Simpson's rule over the entire interval $[a, b]$. Call this approximation *old_approx.*
2. Compute the midpoint x of the interval: $x = (b + a)/2$.
3. Approximate a new interval by applying Simpson's rule to the subintervals $[a, x]$ and $[x, b]$ and add the two results. Call this approximation *new_approx.*
4. If the absolute value of the difference between *new_approx* and *old_approx* is within some limit *sigma* then return *new_approx.*
5. Otherwise, apply the adaptive strategy recursively to each of the subintervals, add the two results, and return the sum as the answer.

Write a Cilk program that contains the function

```
double adaptive_strategy(double (*f)(double),double a,double b,double sigma);
```

Make sure `adaptive_strategy` takes advantage of the parallelism available in the algorithm. Try integrating several functions while varying the *sigma* parameter.

Part d:

How do the algorithms vary in complexity? Compare the accuracy of the answers. What factors affect the execution time and efficiency of the two strategies?

Problem 2 — Hold That Line

In this problem, you will use Cilk to write a search algorithm to solve the game "Hold That Line".

In his *Gamut of Games*, Sid Sackson gives the rules for a very simple 2-player game called "Hold That Line" (which we quote in modified form in the next couple of paragraphs). The game is played on an n by n grid of dots. The first player connects as many dots as he wished in a straight line. For our purposes, lines must begin and end on a dot and must pass through an adjacent point—so there are at most eight directions a line can proceed from a given point. From either end of the first line the second player draws another straight line. From either free end the first player now makes a line. The completed line must be continuous, with no branches, no crossings, and no dot visited twice. Continue until no further lines can be drawn. The player who was forced to make the last line is the loser!

Your task is to list the winning opening moves for the first player in the game. Moves should be of the form (x1, y1), (x2, y2), where the given numbers are the coordinates of the endpoints of the lines involved, and the top left of the board is (0,0). An opening move is a winning move if the first player can always respond to the second player's subsequent moves in such a way that the first player will win the game—that is, the first player has a winning strategy from the opening move.

We consider opening moves to be distinct if they defer by rotation or reflection. If you wish to account for symmetry in your solution (it will make it run much faster!) by all means do so, but remember to account for this in your solution.

If there are no winning opening moves, you program should print "none". Note that there are no winning openings when $n = 2$, for example.

Part a:

Now, write a C version of a program to output the winning openings for n by n Hold That Line given n as a command line argument. Remember, you should consider rotations, reflections, etc. of solutions you have already found to be distinct solutions to the problem—this means that you can get away with fairly naive code to do the computation.

This is essentially a generate-and-test problem—generate opening moves and show that they are winning. Cleverness is called for. You might try the following:

- Note that the only board states which matter are the visited points, the occupied edges (from which the visited points can be derived), and the open endpoints. Player to move is important, but can be inverted pretty easily.

- Any winning opening move involving three or more points automatically means that a smaller opening move is a losing move (as the second player can extend the smaller move to the larger one).
- You will want to store partial solutions, but be careful—you can't store the board state exhaustively.
- Games will develop *unreachable regions*. You can safely ignore the exact pattern of connectivity (or lack thereof) for points in an unreachable region.
- It may help to consider the dual game, in which players take turns erasing line segments from a final position.
- You can sink arbitrary amounts of time into optimizing your solution; don't get too fancy unless you plan on participating in the Challenge!

You should be writing your original C program with an eye to parallelizing it. You want to be able to write and debug this sequential version completely, and then turn it into a Cilk version with minimal changes. Thus you won't necessarily be writing the most efficient possible *sequential* solution to the problem.

When your C code is debugged and runs correctly, you can move on to the next part. There's nothing to turn in for this part; keep your solution, though, in case you get carried away later on.

Part b:

Now that you have a working C program, you should create an equivalent Cilk program by adding appropriate `spawn` constructs. Try to keep your first attempt as simple as possible; you will be better off using special features of Cilk only after you have gotten your basic code running.

Part c:

Run your Cilk program for $n = 3$, and with one, two, four and eight processes. What was the critical path length and wall-clock time in each case (you needn't be rigorous here)? Did your run time decrease linearly with more processes? Why or why not?

Part d:

Challenge (optional): How fast can you get your Cilk code to run in parallel? We'll be looking for programs with the best wall clock times and the shortest critical paths on the Athena Suns using eight processes. Here's the place to play around with the symmetries of the problem! Past years have yielded the magic squares code found in the Cilk distribution.