CS 240A Assignment 3:
Cilkified Inner Products

Assigned April 18, 2011

Due by 11:59 pm Monday, April 25

The purpose of this assignment is to gain familiarity with Cilk++ constructs and tools, and to think about different ways of parallelizing an algorithm using Cilk++. You will write a parallel routine that computes the inner product (dot product) of two vectors in three different ways. Your goal is to compare, understand, and optimize the performance of this simple computation.

As usual, you'll do this in a group of two from different departments, and after it's due you'll swap submissions with another team to write a review.

1. Background

The inner product of two vectors is the sum of their elementwise multiplications. In pure C, the inner product of two real *n*-vectors can be implemented as follows:

```
double innerprod = 0;
for(int i=0; i< n; ++i)
{
	innerprod += a[i] * b[i];
}
```

In C++, the standard library has a dedicated function for computing the inner product. It is made available by including the <numeric> header. For the purposes of this assignment (and this class), you don't have to know and use C++, but you're allowed to do so because the Cilk++ compiler is an extension of C++ and will accept all C++ constructs.

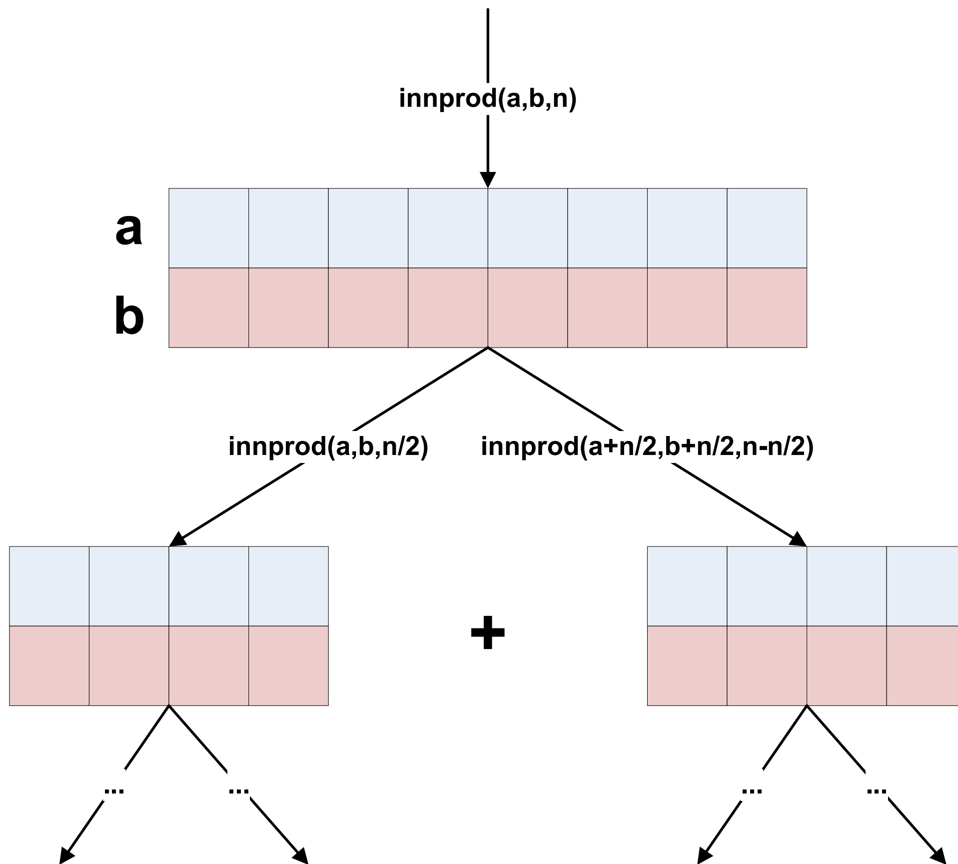
```
int innerprod = std::inner_product(a, a+n, b, 0);
```

2. What to implement

You will write three different functions to evaluate the inner product in parallel.

1. *rec_cilkified(double * a, double * b, int n)*

   This will evaluate the inner product recursively, by splitting each array into two at each stage of the recursion and adding the partial sums at the end. You should switch to a sequential execution when the subarray sizes become smaller than a threshold called *coarseness*. Here is a picture of the calculation:



2. loop_cilkified(double * a, double * b, int n)

   This function contains two nested loops: the outer loop executes *n / coarseness* times, and the inner loop executes *coarseness* times. You should parallelize the outer loop with *cilk_for,* and let each inner loop execution proceed sequentially. Finally, you'll combine the results of the inner loop executions by adding them all together sequentially.

3. hyperobject_cilkified(double * a, double * b, int n)

   For this function, you'll use a hyperobject. Specifically, you'll use a reducer similar to one shown in page 54 of the Cilk++ programmer's guide, and let the Cilk++ reducer concept take care of data races and combining the results.

There is a driver / harness code called "innerproduct.cilk" under hw3 on the course web site. You can use this file as a template and implement the required functions, which are left blank for you to fill in. The same directory contains a sample makefile and a header with the timing function.

Your program will be executed as:

>> **./innerproduct [sizeofarray]**

where the parameter is optional and default is 1 million.

If the harness reports "incorrect" on your results, use **cilkscreen** to identify any data races that might exist in your code. (Don't forget that you must build the debug version of your code for **cilkscreen** to report line numbers where data races occur.)

You can use any multicore shared-memory machine you want for this homework. The course web page has a pointer to download the Cilk++ system for any Intel-architecture machine. I suggest that you debug on your own machine or on CSIL (which has Cilk++ installed, and dual-core machines), and then do performance runs on Triton.

3. What to report

For all your experiments, you should just time the actual inner-product computation, not the input parsing, data setup, and output.

1. Run your code with different input sizes (**sizeofarray** = 10^4,10^5,10^6,10^7,10^8) on a fixed number of cores (**CILK_NPROC** = 4). Plot a graph that has input size on its x-axis (in log scale) and parallel efficiency on the y-axis. Put all three lines (one for each parallel routine you wrote) on the same graph.

2. Run your code on different numbers of cores with a fixed input size. (You can use Triton's "PDAF" nodes to get up to 32 cores; see Stefan's documentation.) Plot a graph that has the number of cores on its x-axis (in normal scale) and parallel efficiency on the y-axis. Plot all three lines on the same graph.

3. Experiment with different *coarseness* values and report on the sensitivity of the performance to different values.

4. Using what you've learned so far, make the best-performing dot-product code you can. It is likely that your code won't enjoy linear speedup. What might be the reason? Do you think that the algorithm does not have sufficient parallelism, or do you think there is another bottleneck other than parallelism?

Using **cilkview**, you can empirically find out the parallelism of your code. However, you should remember that the tool estimates parallelism for the entire program, meaning that your possibly sequential input parsing will make your program look less parallel than it actually is. The Cilk++ programming guide explains how to estimate work and span for a section of code.