Performance Engineering of Software Systems October 21, 2010
Massachusetts Institute of Technology 6.172
Professors Saman Amarasinghe and Charles E. Leiserson Handout 10

Parallel Programming with Cilk I

Last Updated: October 26, 2010

In this project you will learn how to parallel program with Cilk. In the first part of the project, you will be introduced to the Cilk parallel programming platform and learn how to use the Cilkview and Cilkscreen tools. Project 4.1 is to be done indvidually. (The next part, Project 4.2, will be done in groups of two.)

Please answer the questions in this handout and submit an *individual* writeup on Stellar. There is no code to be submitted for this part of the project.

Getting the Code

Use the following command to clone the git repository containing the project 4.1 code:

```
git clone /afs/csail/proj/courses/6.172/student-repos/project4.1/ project4.1
```

This should check out three directories: `qsort`, `mm`, and `nbodies`.

Helpful Hints

- When running a Cilk program, you may specify the number of worker threads to `N` using the `-cilk_set_worker_count=N` command line parameter (defaults to the number of cores on the system).
- When using Cilkview or Cilkscreen, you should run your programs on smaller inputs than when simply running them standalone since there is a large performance penalty for instrumentation.

1 Quicksort

In this problem, you will experiment with an existing Cilk++ project, and learn how to use the Cilkview Performance Analyzer and Cilkscreen Race Detector. You should make running your Cilk applications through Cilkview and Cilkscreen a standard practice.

1.1

Build the qsort binary by running make. This will produce a parallel quicksort binary. The binary takes two optional arguments. The first specifies the number of data points (defaults to 10 million), and the second specifies the number of trials to run (defaults to 1).

Run quicksort with the 50,000,000 data points (the default) using 1 through 20 threads using the Cilkview tool:

```
pnqsub $LOCKER64/cilkview -trials all 20 ./qsort.64
```

What are the execution times? What happens when you run `qsort` with more threads than there are processors on the cloud machines (12)?

When running Cilkview directly on the cloud machines (rather than through PNQ), a speedup graph is automatically displayed with parallelism bounds and measured speedup. You may try this with:

```
cilkview -trials all ./qsort.64
```

(You must have an X server running on your local machine and have X11 forwarding enabled on your SSH connection to view the graph on your local machine.)

When submitting CilkView jobs over the PNQ batch system, the results are saved to a file, so you can load the results into `gnuplot` manually to view the results. To do this, run `gnuplot` and enter `load "qsort.plt"`. (Again, X must be configured properly.)

1.2

Uncomment the following line near the top of the file to introduce a race condition in the parallel code:

```
#define INTENTIONAL_RACE
```

Look at the code enabled by this change and explain how the race could cause quicksort to fail to sort the array of integers.

1.3

Run `qsort` through Cilkscreen using the following command:

```
pnqsub $LOCKER64/cilkscreen ./qsort.64 10000
```

or

```
cilkscreen ./qsort.64 10000
```

Is Cilkscreen able to detect the race? Obtain the approximate failure rate when sorting 10000 integers with 12 threads. Run `qsort.64` with at least 1000 trials to obtain the failure rate.

2 Matrix Multiplication

In this part, you will write a multithreaded program in Cilk++ to implement matrix multiplication. One of the goals of this assignment is for you to get a feeling of how ***work***, ***span***, and ***parallelism*** affect performance. First, you will parallelize a program that performs matrix multiplication using three nested loops. Then, you will write a serial program to perform matrix multiplication by divide-and-conquer and parallelize it by inserting Cilk keywords.

For those of you who have not looked at matrix multiplication in a little while, the problem is to compute the matrix product

$$C = AB ,$$

where C, A, and B are $n \times n$ matrices. Each element c_{ij} of the product C can be computed by multiplying each element a_{ik} of row i in A by the corresponding element b_{kj} in column j in B, and then summing the results, that is,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

For more information on matrix multiplication, please see `http://en.wikipedia.org/wiki/Matrix_multiplication#Ordinary_matrix_product`.

The nested-loop and divide-and-conquer versions of these programs can be adapted to work with arbitrary rectangular matrices. To simplify the interface, however, we limit ourselves to $n \times n$ square matrices.

Matrix multiplication using loop parallelism

The file `mm_loops.cilk` contains two copies of a $\Theta(n^3)$-work matrix multiplication algorithm using a triply nested loop. The first copy (`mm_loop_serial`) is the control for verifying your results — leave it unchanged. The second copy (`mm_loop_parallel`) is the one that you will parallelize. This file also contains a test program that verifies the results of your parallel implementation and also provides infrastructure for timing and measuring parallelism.

2.1

Compile `mm_loops` with optimization, and verify that it operates correctly. Supply the `--verify` command-line option to force running all tests.

Now parallelize the `mm_loop_parallel` function by changing the outermost `for` loop into a `cilk_for` loop. Verify correct results with the `--verify` option. Run Cilkview on your program and report the theoretical and actual speedup. Do not use the `--verify` option when running Cilkview.

2.2

Change the outermost `cilk_for` back into a serial `for` loop and change the middle `for` loop into a `cilk_for` loop. Repeat the test with the `--verify` option, and then report the results from Cilkview. Did any results change? Try making both loops parallel. Which of these combinations produces the best results?

Matrix multiplication by divide-and-conquer

Divide-and-conquer algorithms often run faster than looping algorithms, because they exploit the microprocessor cache hierarchy more effectively. This section asks you to write a divide-and-conquer implementation of matrix multiplication. You will find the source code for the incomplete program in `mm_recursive.cilk`. The program contains two implementations of matrix multiplication. The `mm_loop_serial` function is the same as before and is provided for verification and timing comparisons. The `mm_recursive_parallel` function is the skeleton of a divide-and-conquer implementation.

Your recursive implementation will be based on the identity

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

where A_{11}, A_{12}, etc., are submatrices of A. In other words, matrix multiplication can be performed by subdividing each matrix into four parts, then treating each part as a single element and (recursively) performing matrix multiplication on these partitioned matrices. (The number of columns in A_{11} must match the number of rows in B_{11}, and so forth.) Although the algorithm operates recursively, its work is still $\Theta(n^3)$, the same as the straightforward algorithm that employs triply nested loops.

2.3

Compile `mm_recursive`, and verify that it compiles but fails to run successfully when using the `--verify` command-line argument. The failure is caused by the fact that the `mm_recursive_parallel` has not been fully implemented yet.

2.4

In the file `mm_recursive.cilk`, fill in code in the `mm_internal` function to implement the divide-and-conquer algorithm. The error-prone task of subdividing the matrices into four parts has been done for you. All you need to do is to fill in the recursive calls (eight in total – one for each of the eight matrix-multiplications in the algorithm). Compile and run your new `mm_recursive` program and verify that it runs successfully.

2.5

Which recursive calls to `mm_internal` may be legally executed in parallel with one another and why?

Make your recursive function parallel by adding the `cilk_spawn` keyword in front of some of the recursive calls. You will need to add calls to `cilk_sync` as well in order to separate recursive calls that would otherwise cause a data race and to ensure that all of the work is complete before returning from the function.

Compile and run your new `mm_recursive` program. Verify that it is correct and report the results given by Cilkview. For large matrices, how does the performance of the recursive algorithm compare with the nested-loops algorithm on a single processor?

2.6

Uncomment the line near the top of the program that reads:

`#define USE_LOOPS_FOR_SMALL_MATRICES`".

This change causes the algorithm to change from divide-and-conquer recursion to a triply nested loop for small matrices. How does this change impact performance? How does the performance of this new version compare to the previous versions?

3 N-Bodies Simulation

In this part, you will modify a program that simulates a set of planetary bodies drifting in space in the neighborhood of a single, massive "sun." Each body b_i (including the sun) has a mass m_i and an initial velocity v_i, and is attracted to another body b_j of mass m_j by a force f_{ij}, which obeys the formula for gravitational attraction,

$$f_{ij} = G\frac{m_i m_j}{r^2} \ ,$$

where G is the gravitational constant, and r is the distance between the bodies. The force on b_i is directional and pulls b_i towards b_j. The total force f_i on body b_i is the vector sum of the forces f_{ij} for all $j \neq i$.

The N-bodies simulation program begins by creating N bodies (the number N is specified on the command line) with randomly-selected masses in a random distribution around the sun. All of the bodies are given initial velocities such that the entire system appears to have a clockwise spin. A two-dimensional coordinate system is used for simplicity.

The simulation progresses by computing successive positions of each of the *N* bodies for successive moments in time using a two-pass algorithm. The first pass computes the force on each body as the sum of the forces exerted on it by all of the other bodies in the system. The second pass adjusts the velocity of each body according to the force computed in the first pass and moves the body's position according to its average velocity during that quantum of time. Every few time quanta, a snapshot of the entire system is rendered as a picture in `png` format. You can view the result as a short movie using the provided JavaScript-powered web page. After running the `nbodies` binary, type `make publish` to copy the `nbodies.html` file and your PNG outputs to your CSAIL webpage. Then visit `http://people.csail.mit.edu/<username>/nbodies/` to view your movie.

Note that the execution times for these programs can be fairly long. Since Cilkview runs the program at least 5 times, expect to wait several minutes for a Cilkview run to complete. You may wish to reduce the number of images produced to save time. If you choose to do so, do not reduce the number of bodies in the simulation, or you will reduce the total parallelism in your program.

When running Cilkscreen to detect races, you can and should reduce the number of bodies and images produced to a bare minimum (e.g. 10 bodies and 2 images). Note that if you produce fewer images the visualization will be truncated, and the web page may display some parts of the previous run if there were leftover PNG files in your directory.

Burdened parallelism in the N-bodies simulation

The file `nbodies_loops.cilk` implements the n-body simulation. The `calculate_forces` function uses a pair of nested loops to compute the forces for every pair of bodies. The `update_positions` function uses a single loop to update the position of each body.

The `nbodies` binary accepts two arguments: the first is the number of bodies in the simulation (defaults to 800), and the second is the number of PNG frames to produce (defaults to 100). The number of simulation steps computed between frames is 40. All of these defaults and constants may be changed by modifying the `#defines` at the top of the `common.h` file.

3.1

Compile `nbodies_loops` with optimization, and run it with the default number of bodies (800). View the simulation in a web browser by running `make publish` and visiting:

`http://people.csail.mit.edu/<username>/nbodies/`.

Click the "Start" button to view the movie. (JavaScript must be enabled.) Parallelize the program by changing the loop in `update_positions` and the inner loop in `calculate_forces` to `cilk_for`. Run the program in Cilkview and report the results.

3.2

Insert the line "`#pragma cilk_grainsize=1`" just before the `cilk_for` in `calculate_forces`. Run the program in Cilkview and compare its performance with the previous run. Explain these results.

3.3

Unlike in the matrix-multiplication example, you cannot parallelize the outer loop in `calculate_forces`, because doing so would cause multiple iterations to race on updating a single body's forces. (Try it in Cilkscreen if you want.) However, you can invert the inner and outer loops of your current code so that the `cilk_for` loop over i is on the outside. Try it, and report your results from Cilkview. (You should remove the `grainsize` pragma.) Which version is faster and why?

Resolving races with locks

The formula for computing the gravitational force between two bodies b_i and b_j is symmetrical such that $f_{ji} = -f_{ij}$ (i.e., the magnitude of the force on both bodies is the same, but the direction is reversed.) Our current implementation of the N-bodies simulation, however, computes each force twice: once when computing the force that b_j applies to b_i, and again when computing the force that b_i applies to b_j. We can halve the total number of iterations in `calculate_forces` if we take advantage of this fact and compute the force only once for each pair of bodies.

In the file `nbodies_symmetric.cilk`, we have modified the inner loop in `calculate_forces` to avoid calculating forces that have already been computed in an earlier iteration (according to the serial ordering). When computing a force f_{ij} and adding it to f_i, we also compute the inverse force $_{ji}$ and add it to f_j. Unfortunately, this simple optimization has its problems, as we shall see.

3.4

In `nbodies_symmetric`, parallelize `update_positions` and the outer (i) loop of `calculate_forces`. Compile `nbodies_symmetric` and run it with a command-line argument of 800. Did we see the expected speedup of 2 versus the (parallel) version of `nbodies_loops`? Run the program again in the Cilkscreen race detector, but shorten the run time by using a command-line argument of 10 instead of 800 (and you shouldn't have to use PNQ for this). Where did the races come from, and why weren't they visible in the initial run?

3.5

One way to fix the race is to use a *mutex* (mutual exclusion) lock to mediate concurrent access to each object. Add a member `mtx`, of type `cilk::mutex` to the `Body` struct. In `add_forces`, insert the statement "`b->mtx.lock();`" before updating `b->xf` and `b->yf` and insert "`b->mtx.unlock();`" after updating them. Run the program with Cilkview (using the original command-line argument of 800) and report the theoretical and actual speedup.

Solving races without locks

For sufficiently large data sets, the previous solution should produce little lock contention. Nevertheless, both locks and atomic instructions are expensive, even in the absence of contention, because they interrupt the CPUs' pipelines and force serializing operations that the CPU would have internally performed in parallel. It would be ideal if we could parallelize the N-bodies problem without introducing data races at all, thus eliminating the need for locks or atomic instructions.

One such solution, due to Matteo Frigo, uses divide-and-conquer parallelism in a way that ensures that no two parallel strands attempt to modify the same body. The algorithm, with the Cilk keywords removed, is implemented in the file `nbodies_nolocks.cilk`.

Figure 1 shows the core of the program. The lines labeled $[A]$ (Lines 44–45) can be executed in parallel with each other. Similarly, the lines labeled $[B]$ (lines 16–17) can be executed in parallel with each other, and the lines labeled $[C]$ (lines 18–19) can be executed in parallel with each other. This program is not meant to be obvious. Let's explore what it does.

The serial program is equivalent to calling `add_force(&bodies[i], fx, fy);` and `add_force(&bodies[j], fx, fy);` for all $0 \leq i \leq j < N$. Another way to look at it is that a plot of the points (i, j) such that $0 \leq i \leq j < N$ comprise the shaded area shown in Figure 2.

Procedure `triangle` traverses this triangle in parallel, and in fact it is a little bit more general, because it traverses any triangle of the form $n0 \leq i \leq j < n1$. Initially, we set $n0 = 0$ and $n1 = N$ in `cilk_main`.

Procedure `triangle` works by recursively partitioning the triangle. If the triangle consists of only one point, then it visits the point $(n0, n0)$ directly. Otherwise, the procedure cuts the triangle into one rectangle and two triangles, as shown in Figure 3.

The two smaller triangles can be executed in parallel, because one consists only of points (i, j) such that $i < nm$ and $j < nm$, and the other consists only of points (i, j) such that $i \geq nm$ and $j \geq nm$. Thus, the two triangles update nonoverlapping regions of the `force` array, and thus they do not race with each other. However, the rectangle races with both triangles, and thus we need a `cilk_sync` statement before processing the rectangle.

To traverse a rectangle we use procedure `rect`, which also works recursively. Specifically, if the rectangle is large enough, the procedure cuts the rectangle $i_0 \leq i < i_1$, $j_0 \leq j < j_1$, into four smaller subrectangles, as shown in Figure 4.

The amazing thing is that the two black subrectangles can be traversed in parallel with each other without races. Similarly, the two gray subrectangles can be traversed in parallel with each other without races. Since the black and gray subrectangles race with each other, however, we must use a `cilk_sync` statement after processing the first pair of subrectangles.

To see why there are no races between the two black subrectangles (the same argument applies to the gray) observe that the i-ranges of the two subrectangles do not overlap, because one is smaller than i_m and the other is larger. For the same reason, neither do the j-ranges overlap. In order for races not to occur, however, we must also prove that the i-range of one subrectangle does not overlap with the j-range of the other, because we are updating both `bi` and `bj`. This property holds because when `triangle` calls `rect` initially, the i-range is $n0 \leq i < nm$, whereas the j-range is $nm \leq j < n1$, so the two ranges never overlap.

This algorithm partitions the original data into smaller and smaller subsets. Thus, in addition to avoiding races and locks, the algorithm exploits cache locality in a way similar to that of the recursive matrix-multiplication example.

3.6

The file `nbodies_norace.cilk` implements the divide-and-conquer algorithm as a serial program. Notice that the `rect` routine coarsens the recursion and reverts to a looping implementation when the rectangle is sufficiently small (one of the sides has length at most `THRESHOLD`). The `triangle` routine is not similarly coarsened, however. Explain why coarsening `triangle` would not improve performance significantly for large problem sizes.

```
// update the force vectors on bi and bj exerted on each by the other.
void add_force(Body* b, double fx, double fy)
{
    b->xf += fx;
    b->yf += fy;
}

/* traverse the rectangle i0 <= i < i1,  j0 <= j < j1 */
void rect(int i0, int i1, int j0, int j1, Body *bodies)
{
    int di = i1 - i0, dj = j1 - j0;
    const int THRESHOLD = 16;
    if (di > THRESHOLD && dj > THRESHOLD) {
        int im = i0 + di / 2;
        int jm = j0 + dj / 2;
        rect(i0, im, j0, jm, bodies);  // [B]
        rect(im, i1, jm, j1, bodies);  // [B]
        rect(i0, im, jm, j1, bodies);  // [C]
        rect(im, i1, j0, jm, bodies);  // [C]
    }
    else {
        for (int i = i0; i < i1; ++i) {
            for (int j = j0; j < j1; ++j) {
                // update the force vector on bodies[i] exerted by bodies[j]
                // and, symmetrically, the force vector on bodies[j] exerted
                // by bodies[i].
                if (i == j) continue;

                double fx, fy;
                calculate_force(&fx, &fy, bodies[i], bodies[j]);
                add_force(&bodies[i], fx, fy);
                add_force(&bodies[j], -fx, -fy);
            }
        }
    }
}

// traverse the triangle n0 <= i <= j < n1
void triangle(int n0, int n1, Body *bodies)
{
    int dn = n1 - n0;
    if (dn > 1) {
        int nm = n0 + dn / 2;
        triangle(n0, nm, bodies);  // [A]
        triangle(nm, n1, bodies);  // [A]
        rect(n0, nm, nm, n1, bodies);
    }
    else if (dn == 1) {
        // Do nothing.  A single body has no interaction with itself.
    }
}

void calculate_forces(int nbodies, Body *bodies) {
    triangle(0, nbodies, bodies);
}
```

Figure 1: A lock-free code for N-bodies.

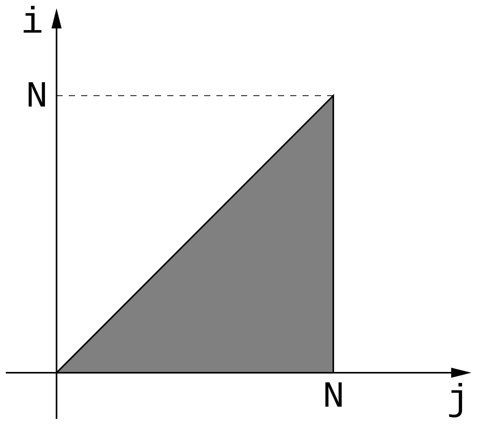

Figure 2: Traversing the space, $0 \le i \le j < N$

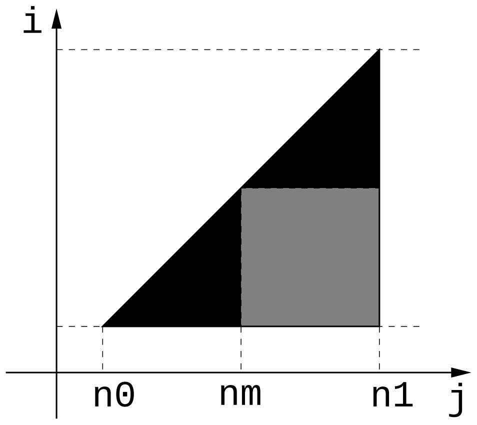

Figure 3: Cutting a triangle into two smaller triangles and a rectangle

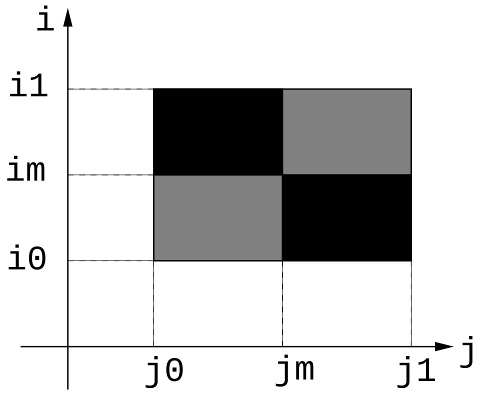

Figure 4: Dividing a rectangle into four

3.7

Add `cilk_spawn` and `cilk_sync` statements to implement the divide-and-conquer parallel algorithm. Briefly describe the changes you made. Also add a `cilk_for` to parallelize the `update_positions` function.

3.8

Confirm that the program is race free by running it in the race detector (use only 10 bodies to avoid long run times). Run the program in Cilkview and report the results. Compare the performance to runs of earlier versions of the program.