Performance Engineering of Software Systems September 28, 2021
Massachusetts Institute of Technology 6.172
Saman Amarasinghe, Charles E. Leiserson, and Jonathan Ragan-Kelley Handout 6

Homework 4: Cilk Primer

Due: 11:59 P.M. (ET) on Tuesday, October 5th, 2021

Last Updated: September 29, 2021

In this homework you will experiment with parallelizing programs with Cilk. You will learn how to use Cilksan to detect and solve determinacy races in your multithreaded code, and how to measure a program's parallelism using the Cilkscale scalability analyzer.

Contents

1 Getting started

Getting the code

You can get this assignment's code using Git:

```
$ git clone git@github.mit.edu:6172-fall21/homework4_<username>.git homework4
```

Submitting your solutions

Submit your write-up on Gradescope and your code via Git before the deadline stated at the top of this handout. For each question we ask, give a short response of 1–3 sentences. Paste program outputs where necessary.

Reporting performance

To test the performance of parallel code, you should use **`awsrun8`**, a variant of `awsrun` that submits jobs to cloud machines with 8 cores. It's important to use `awsrun8` for all performance measurements involving parallel code.

2 Recitation: Parallelism and race detection using Cilk

Please answer the checkoff questions on your own and show your responses to a TA after you have completed *all* checkoffs in this section.

2.1 Introduction to Cilk

The `cilk_spawn` and `cilk_sync` keywords

Compile the `fib` program in the `fib/` subdirectory. Then, time the execution for finding `fib(45)` using the `time` command:

```
$ awsrun8 time ./fib 45
```

You will get 3 different times as outputs, labeled `real`, `user`, and `sys`. The `real` time is wall-clock time, which is the total elapsed time from beginning to end of the execution. The `user` time is the amount of time a CPU spent in user mode. The `sys` time is the amount of time a CPU spent in the kernel. The sum of `user` and `sys` is the actual CPU time of the command. Since the current `fib` is a serial program, you will find that wall-clock time is slightly higher than CPU time.

Next, we want to parallelize the program to take advantage of the other 7 processor cores on the `awsrun` machines. You can do this by adding `cilk_spawn` in front of function calls that you want to execute in parallel. You also need to add `cilk_sync` to wait for all spawning tasks to finish. These keywords can be used when the Cilk header is included:

```
#include <cilk/cilk.h>
```

You can specify the number of Cilk workers that execute a program by setting the environment variable `CILK_NWORKERS`. You can, for example, set it to 8 workers when running a program by prepending `CILK_NWORKERS=8` to the command you wish to run. If you do not specify it, the number of workers will default to the number of cores in the system where the program is being run. Setting the number of Cilk workers is especially useful for evaluating execution time as a function of workers. For example, you may run `./fib` with 4 workers on the 8 core AWS runner with the following command:

```
$ awsrun8 CILK_NWORKERS=4 ./fib 45
```

Checkoff Item 1: Parallelize `fib` using `cilk_spawn` and `cilk_sync`, and report the execution time when using 1, 4, and 8 Cilk workers.

You might find that the new version is not faster than the first one (in terms of `real` time). Furthermore, the CPU time is much higher than wall-clock time, because the program uses multiple processors to run. Under what circumstances would the parallel version be slower? Try to fix your program using "coarsening" to get a parallel speedup.

Checkoff Item 2: Describe your approach to coarsening the program, and report your parallel execution time for the coarsened version of `fib` on 1, 4, and 8 Cilk workers.

The `cilk_for` keyword

The `transpose/` subdirectory contains the source code for `transpose`, an in-place matrix-transpose program. As you saw in lecture, you can replace the `for`-loops with `cilk_for`-loops to parallelize `transpose`.

Checkoff Item 3: Parallelize `transpose` using `cilk_for`, and report your execution time with input size 10000 for 1, 4, and 8 Cilk workers.

2.2 The Cilksan race detector

The Cilksan race detector allows you to check whether your Cilk program has a determinacy race. It provides detailed output that specifies the line numbers of two memory accesses, often a read and a write, that were involved in the race.

Compile and run the `qsort-race` program in the `qsort-race/` subdirectory. This code was parallelized by naively adding `cilk_spawn` and `cilk_sync` to a serial quicksort program. This code has a race!

Before you run Cilksan, take a look at the quicksort code and see if you can identify the determinacy race. See if you can expose the race condition by running a few tests on `awsrun8`, too. Don't be discouraged if you are unable to expose the race by running tests — but also do not allow yourself to be fooled! This code does, indeed, have a race. Like many subtle determinacy races, the one present in quicksort is difficult to elicit and identify without the use of tools.

We can use Cilksan to detect the race by recompiling the code as follows:

```
$ make -B CILKSAN=1
```

The `-B` flag tells `make` to rebuild all targets. It is practically the same as running `make clean` right before.

When running the newly compiled binary, Cilksan outputs its report to `stderr`. We can now expose this race on a fairly small input of 10 elements:

```
$ ./qsort-race 10 1
```

Checkoff Item 4: Use Cilksan to find this race. Then, fix the race and use Cilksan to confirm that no more races exist. Report the line numbers in the code where the read/write race occurs, and give a brief description of what was happening to cause this race.

2.3 The Cilkscale scalability analyzer

The `qsort/` subdirectory contains `qsort`, another parallel quicksort program. This version of quicksort should not have any races, but you should of course verify this by using Cilksan.

You can use Cilkscale to analyze the scalability of this quicksort program. We can build `qsort` with Cilkscale as follows:

```
$ make -B CILKSCALE=1
```

Since Cilkscale uses timing measurements to compute parallelism, it is usually a good idea to run it on a quiesced machine — such as those provided in the AWS job queue. For this assignment, however, we recommend you run Cilkscale locally on your VM instead of using `awsrun8` to avoid clogging the queue.

Checkoff Item 5: Report the parallelism computed by Cilkscale on the quicksort program for a few different sized inputs.

Cilkscale's command-line output includes work and span measurements for the Cilk program in terms of empirically measured times. Parallelism measurements are derived from these times.

For some programs, such as `fib`, there may be some variability in the reported parallelism numbers.

Checkoff: Explain your responses to Checkoff Items 1–5 to a TA. Hop in the queue and read the following explanation while you wait.

A simple struct `wsp_t` contains the number of nanoseconds for work and span. This data is collected immediately before and after the `sample_qsort()` execution at lines 124 and 126 of `qsort.c`. Then these two measurements are subtracted and dumped (in line 134) to `stdout` in CSV format, with the first column being the label of the measurement. At the end, the same measurements are output for the program as a whole with an empty label. The final measurement includes all the setup and teardown code, which pollutes the measurement we are interested in. Because the dump to `stdout` can interleave with other program output, you might want to set the environment variable `CILKSCALE_OUT=<filename>.csv` to redirect Cilkscale output to a specific file (you will only be able to access that file when running Cilkscale instrumented programs locally — `awsrun` currently doesn't return output files).

In addition to a span column, you are also seeing a "burdened span" column. Burdened span accounts for the worst possible migration overhead, which can come from work-stealing and other factors. If you are interested in how Cilkscale works, you can check out the paper on Cilkview, Cilkscale's predecessor.

3 Homework: Parallelizing Floyd-Warshall

The all-pairs shortest path problem is a classic algorithms problem. The goal is to compute the minimum distance between every pair of vertices in a graph, where distance is defined as the sum of edge weights in a path that connects a pair of vertices. A standard algorithm for solving this is known as Floyd-Warshall, which initializes a distance matrix with the weighted graph adjacency matrix and iteratively "relaxes" its edges. If you don't recall this algorithm, now is a good time to pull out your handy copy of *Introduction to Algorithms* by CLRS.[1]

The relaxation step of Floyd-Warshall is given by:

$$D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$$

for a triple (i,j,k) of vertex indices, where D is a matrix containing the currently best known pairwise distances. Essentially, the relaxation step checks if the current best path from vertex i to j is shorter than by going through some intermediate point k. The Floyd-Warshall algorithm is given by the triple loop around this relaxation step as shown in the following pseudocode:

[1] Recall any of the authors?

```
// initially, "A" is the weighted adjacency matrix of the graph
// (distances between unconnected vertices are initialized to +INF)
for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
        }
    }
}
```

And that's it!

The next natural step is to construct a parallelized version of Floyd-Warshall. The simplicity of the algorithm, in addition to the power of `cilk_for`, make it a tempting target for parallelization. For the rest of this homework, assume that there are no negative-weight cycles.

Enter the `floyd/` subdirectory, and run `make floyd_w`. Executing `./floyd_w -p` just runs your parallelized code. To check correctness, run `awsrun8 ./floyd_w -p -c`.

Write-up 1: Try replacing each of the `for`-loops separately in `floyd_parallel` with a `cilk_for`. For which of the loops does `awsrun8 ./floyd_w -p -c` fail correctness?

Write-up 2: For which of the `for`-loops, parallelized with `cilk_for` and compiled with `CILKSAN=1`, does Cilksan report a race in the program? Use `./floyd_w -p -m 1 -n 10` when running your program for each loop.

You might notice that the inner loop does in fact have a race condition, but Cilksan does not seem to report it. It turns out that the Cilk compiler is coarsening that loop to the point where there are no parallel iterations (If you aren't convinced, try running it with Cilkscale and looking at the parallelism). To prevent the compiler from coarsening the loop, you can use the directive `#pragma cilk grainsize 1`. Place this pragma right above the inner most `for`-loop when you make it a `cilk_for`. Retest correctness, and rerun with Cilksan to confirm that you now see a race reported.

You should find that the answers to the previous two write-ups are not the same. In fact, there are two loops that when parallelized create races, but in such a way that even if a race happens, the validity of the program is not compromised. These are referred to as ***benign races***. Parallelize both loops, and use that code for the remainder of this homework.

Write-up 3: Why does parallelizing the loops that cause the benign race in the floyd_parallel() (with two loops parallelized) still produce the correct result? Why can't we parallelize the outer-most loop? Assume that there are no negative-weight cycles.

Be careful when answering this question. In particular, explain why the races in floyd_parallel() are benign despite the fact that the races among iterations of the loop in the following code are not benign:

```
// Code to compute the minimum of array A
int m = A[0];
cilk_for(int i = 1; i < n; i++) {
    if (A[i] < m) {
        m = A[i];
    }
}
return m;
```

We now want to measure the scalability of floyd_parallel() with Cilkscale. To get a measurement of just floyd_parallel() apart from the rest of the code, Cilkscale provides wsp_getworkspan() to tell Cilkscale when to start and stop monitoring the program, and wsp_dump() to collect the results. Uncomment the calls to these functions around the switch-case statement in test() in floyd_w.c when measuring parallelism of floyd_parallel() with Cilkscale. Don't forget to use #pragma cilk grainsize 1 on each loop since we want to make sure the compiler does not coarsen either loop.

Once you are ready, run your parallelized Floyd-Warshall with Cilkscale using

```
$ make -B CILKSCALE=1 && CILK_NWORKERS=1 ./floyd_w -p -m 700 -n 1
```

Record the running time of your program by running:

```
$ make clean && make && awsrun8 time ./floyd_w -p -m 1000 -n 1
```

Write-up 4: Compare the reported parallelism by Cilkscale and running times of floyd_parallel() with and without the grainsize #pragma's. Explain the differences you see.

The function fw_divide_and_conquer() provides a clever alternative method due to Matteo Frigo (2012) for parallelizing Floyd-Warshall. The algorithm greedily tries to divide-and-conquer whenever possible, partitioning the three-dimensional (i, j, k) iteration space on one of the indices.

Under certain conditions the two recursion steps can only be split serially, otherwise the two recursion steps can be run in parallel.

For example, the divide-and-conquer algorithm, as seen in `fw_divide_and_conquer()`, uses the condition

```
// "im" is the midpoint of [i0,i1)
!overlaps(i0, im, k0, k1) && !overlaps(im, i1, k0, k1)
```

to determine whether the halves determined by a split on `i` can be run in parallel. That is, if the interval [`i0`,`i1`) does not overlap with the interval [`k0`,`k1`), then the code determines that it is safe to run two halves in parallel.

Write-up 5: Explain why splitting the iteration space on the *i*-dimension induces no races if the above condition holds.

Make sure that the `wsp_getworkspan()` calls still surround the `switch-case` in `test.c` and run the divide-and-conquer algorithm with Cilkscale using

```
$ make -B CILKSCALE=1 && CILK_NWORKERS=1 ./floyd_w -d -m 700 -n 1
```

Record the running time of the divide-and-conquer algorithm with

```
$ make clean && make && awsrun8 time ./floyd_w -d -m 1000 -n 1
```

Write-up 6: Compare the parallelism reported by Cilkscale and the running time of the divide-and-conquer algorithm with the results you obtained in Write-up 4 for `floyd_parallel()`. Are they similar? Why or why not?

The divide-and-conquer algorithm uses a constant `BASE_LIMIT` to decide when to fall back to using serial Floyd-Warshall. Set `#define BASE_LIMIT 1`, and again record the running time and parallelism of your updated program.

Write-up 7: Compare the running time and parallelism of the divide-and-conquer algorithm before and after you changed `BASE_LIMIT`. Explain any similarities and differences in these measurements.

Write-up 8: Why do you think the 6.172 course staff recommends that you avoid benign races in your code whenever possible? If performance or other considerations force you to include a benign race in your code, what might an extraordinary software engineer do to mitigate the impact?

On x86-64 machines, naturally aligned loads and stores are atomic up to 64 bits. For example, suppose that a write to a 64-bit location is concurrent with a read to the same location. If the accesses are atomic, the reader obtains either the value originally in the location or the value that the writer wrote. The value it reads is never composed of some bits of each (or an entirely different value altogether). For 128-bit values, however, it could be that the reader sees the high-order 64 bits of the original value and the low-order 64 bits of what the writer wrote.

Write-up 9 (optional): If we change the `typedef` of `DIST` to `__int128` in `floyd_parallel`, are the races still benign or could an incorrect value be produced? Assume that the graph contains no negative-weight cycles. Explain.

Now, assume that the graph contains a negative-weight cycle. Are the races still benign? Will we always obtain the same output as the serial program produces? Explain.