*Performance Engineering of Software Systems*
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 7

# Homework 4: Reducer Hyperobjects

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

## 1 Getting started

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

### Submitting your solutions

For each question we ask give short responses of 1–3 sentences. Paste program outputs where necessary.

### Reporting performance numbers

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 numbers involving parallel code.

## 2 Recitation: Parallelism and race detection using Cilk

Please answer the checkoff questions on your own and show a TA after you have completed all of the questions 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 time, which is what you see from a clock. The `user` time is the time a CPU spent in user mode. The `sys` time is the 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 time is slightly higher than CPU time.

Next, we want to parallelize the program to take advantage of the other 7 processors 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. Lastly, you should include the Cilk header using

```
01 #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 specify this variable globally for all Cilk programs executed in your bash session by running:

```
$ export CILK_NWORKERS=8
```

You may also specify the number of Cilk workers to 8 for a single execution of a program by prepending `CILK_NWORKERS=8` before the command you wish to run. Setting the number of Cilk workers is especially useful when executing a Cilk program when using `awsrun8`. For example, you may execute

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

This command will execute `fib` on the AWS job queue machines using 4 Cilk workers. You can safely ignore errors about not finding the command to set `CILK_NWORKERS` with `awsrun8` such as the following:

```
[Warning] Cannot find path for command: CILK_NWORKERS=8
```

**Checkoff Item 1:** Parallelize `fib` using `cilk_spawn` and `cilk_sync`, and report the runtime 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 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 runtime 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 runtime 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 locations, often a read and a write, that were involved in the race. The Cilksan tool needs the `libsnappy` package, which you can install on your AWS instance with:

```
$ sudo apt-get install libsnappy-dev
```

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 identify without the use of tools.

We can use Cilksan to detect the race as follows:

```
$ make clean; make CILKSAN=1
```

We can 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 have printed the scalability information at the end of `main` in `qsort.c`:

```
02 #ifdef CILKSCALE
03 print_total();
04 #endif
```

We can build `qsort` with Cilkscale as follows:

```
$ make 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 AWS instance instead of using `awsrun`.

**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 cycle counts, as well as parallelism measurements based on the measured work and span. For some programs, such as `fib`, there may be some variability in the reported parallelism numbers.

**Checkoff:** Explain your responses to the previous five Checkoff Items to a TA.

## 3 Homework: $N$ queens problem and reducers

We introduced the $N$ Queens Problem in Lecture 3: Bit Hacks. The problem is to place $N$ queens on a $N \times N$ chessboard so that no queen attacks another (i.e., no two queens in any row, column, or diagonal). Review the slides to get a feel for how the backtracking procedure works.

For this homework, our recursive implementation in `queens.c` (only 12 lines of code!) looks for all possible solutions and appends them to a list. It works for $N = 8$, a standard $8 \times 8$ chessboard. We chose $N = 8$ for two reasons:

1. Tractable number of solutions. $N = 8$ only has 92 distinct solutions. $N = 16$, for example, has 14772512 distinct solutions.

2. Simple board representation. The $8 \times 8$ squares of the chessboard fit into the bits of a `uint64_t`.

Lecture 3 represented the board as 3 bit vectors `down`, `left`, and `right` of size $N$, $2N - 1$, and $2N - 1$, respectively. The implementation does better! It uses the same 3 bit vectors, but of size $N$, $N$, and $N$ only. This result was invented by Tony Lezard and can be found in an email from 1991:

```
Path: gmdzi!unido!mcsun!uknet!slxsys!ibmpcug!mantis!tony
From: to...@mantis.co.uk (Tony Lezard)
Newsgroups: rec.puzzles
Subject: Re: 8 Queens (NO *SPOILER*)
Message-ID: <eeFmBB1w164w@mantis.co.uk>
Date: 18 Nov 91 18:47:49 GMT
References: <1991Nov16.033939.70781@cs.cmu.edu>
Organization: Mantis Consultants, Cambridge. UK.
Lines: 40
redm...@cs.cmu.edu (Redmond English) writes:
> I wrote a little C program a while ago, which after exhaustively testing
> every position with exactly one queen on each rank (taking about 1 minute
> on an 8MHz 68000 machine), came up with 92 solutions.
Erk. Exhaustive testing? Count me out on that. The following program,
written in ANSI C, uses a backtracking algorithm and gives the correct
answer instantaneously:

#include <stdio.h>
void try(int, int, int);
static int count = 0;

void main() {
   try(0,0,0);
   printf("There are \%d solutions.\n", count);
}

void try(int row, int left, int right) {
   int poss, place;
```

```
        if (row == 0xFF) ++count;
        else {
           poss = ~(row|left|right) & 0xFF;
           while (poss != 0) {
              place = poss & -poss;
              try(row|place, (left|place)<<1, (right|place)>>1);
              poss &= ~place;
           }
        }
     }


     ObPuzzle: Generalize the above to n queens.
     --
     Tony Lezard IS to...@mantis.co.uk OR tony\%man...@uknet.ac.uk
     OR EVEN ar...@phx.cam.ac.uk if all else fails. Great! Kept my .sig down to two
     lines!
```

**Write-up 1: (Bonus)** How does the code snippet above work? Why do you only need $N$ bits for each vector?

Our `queens` implementation uses three bit vectors, where 0s represent available squares and 1s represent unavailable squares. Therefore, the `down`, `left`, and `right` start as all 0s and get filled with 1s as the queens are placed.

### 3.1 It's a race, but no one is winning.

Compile and run `/queens`. Record the serial execution time.

Let's see if we can get any speedups from parallelization. In the `queens` function, add `cilk_spawn` and `cilk_sync` keywords to parallelize the recursive calls to `queens`. Compile and run `/queens` with `wsrun 8`.

When running a Cilk program, you may specify the number of worker threads you'd like to use by setting the environment variable `CILK_NWORKERS`. Specifying the number of Cilk workers can be done by adding `export CILK_NWORKERS=W` (where $W$ is an integer greater than 0) in the command line before `./queens`. Most of the time, you won't have to explicitly set this variable because it defaults to the number of cores on the system. A good place to find Cilk-related reference information is cilk.mit.edu.

**Write-up 2:** What happened when you ran `./queens`? Did parallelization do what you expected?

Something went wrong. Let's use the Cilksan tool to detect the problem. Cilksan will take too long to finish with the current build, so comment out the loop which runs `run_queens I` times in the `main` function. **Remember to uncomment the loop before recording performance numbers for the remainder of the homework.**

Now compile with Cilksan. The `clang` option `-fsanitize=cilk` triggers compilation with Cilksan, and you can build with it automatically by running `make CILKSAN=1`. The resulting binary will run with Cilksan code built-in.

**Write-up 3:** Was Cilksan able to detect the problem? Describe the race condition briefly in words, and report the relevant line numbers for the read and write involved in the race.

### 3.2 Fixing the race

Remove `cilk_spawn` and `cilk_sync` keywords from the `queens` function. Before re-parallelizing our code, we want to make any accesses to the list of solutions, `board_list`, thread-safe.

Here's one strategy that doesn't require mutual exclusion locks: instead of passing in `board_list` to the recursive calls to `queens`, we can create a bunch of temporary `BoardList` objects and pass in a different one to each recursive thread. When the threads synchronize, all of the temporary lists will have been filled, and we can concatenate them all together with the original `board_list`.

In `queens.c`, implement

```
05 void merge_lists(BoardList* list1, BoardList* list2);
```

The function should merge `list2` into `list1` and then reset `list2` to be empty. For example, say `list1` has 3 nodes and `list2` has 5 nodes. After a call to `merge_lists(list1, list2)`, `list1` should have 8 nodes and `list2` should be empty. We'll use this call to concatenate several lists together. Remember to handle cases where one or both of the lists are empty and to update all 3 fields of `BoardList`.

**Write-up 4:** What is the most efficient way to concatenate two singly linked lists? What is the asymptotic running time of your `merge_lists` function?

Now, implement the strategy of passing in temporary lists to the recursive calls and merging them at the end. Compile and run `./queens` to make sure your serial code is still correct. If it is, re-parallelize the code with `cilk_spawn` and `cilk_sync`. Run Cilksan to verify that there are no races. Then record the parallel execution time.

> **Write-up 5:** Briefly explain your implementation. How does the parallel code perform compared to the serial code? Try to explain any difference in performance.

Recall that, in Homework 2, you coarsened the recursion of merge sort. Coarsen the recursion of `queens` so that if it hits a base case, then it uses the original implementation. Record the new parallel execution time. **Remember to uncomment the loop before recording performance numbers for the remainder of the homework.**

> **Write-up 6:** What is the base case you used? How does the parallel code with coarsening perform compared to the parallel code without? Try to explain any difference in performance. How does it perform compared to the serial code?

### 3.3 Cilk reducers

#### Background

Cilk provides a unique programming construct called a ***reducer hyperobject***, or ***reducer***. Conceptually, a reducer is a variable that can be safely used by multiple Cilk strands running in parallel. In a parallel execution of a program, the Cilk runtime maintains multiple views of a reducer. The runtime eliminates the possibility of a race by ensuring that each view can be accessed by only one Cilk strand at a time. After all parallel strands have been synchronized (i.e., after a `cilk_sync`), Cilk guarantees that all views of a reducer have been merged together into a single view. In Cilk, reducers are based on "monoids," and thus Cilk guarantees that a parallel execution of a program updates a reducer and merge views in a way that is equivalent to a serial execution of the same program.

More precisely, a ***monoid*** is an algebraic structure on a set of elements $T$ with an associative binary operator $\oplus$ and an identity element $e$. As a simple example, for an integer sum monoid, $T$ is $\mathbf{Z}$ (the set of integers), $\oplus$ is $+$ (integer addition), and the identity is $e = 0$.

The serial execution of a Cilk program typically performs a sequence of updates to a reducer $X$, which correspond to a sequence of $\oplus$ operations with elements $x_i$ drawn from $T$. For example, consider a program with a reducer $X$ with initial value $x_0$, and suppose that a serial execution of the program performs a sequence of updates $x_1, x_2, \ldots, x_7$ to a reducer $X$. The serial execution combines these updates one at a time, i.e., it produces a final value for $X$ of

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7) \; . \tag{1}$$

Because $\oplus$ is associative and has an identity element $e$, however, other ways to combine these updates also produce the same final result. For example, one could also combine updates as follows:

$$((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus (((e \oplus x_5) \oplus x_6) \oplus x_7) \ , \tag{2}$$

and still produce the same result. The original serial execution order in Equation (1) uses a single ***view*** of $X$, updates occur sequentially. In contrast, the execution represented by Equation (2) uses two views, one for value $(x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4)$, and one for the value $(x_5 \oplus x_6 \oplus x_7)$. Having two views allows two processors to update $X$ in parallel without races.

More generally, a parallel execution may create more reducer views and combine them in more complicated but equivalent ways. For example, Equation (3) uses three views, and Equation (4) uses five views.

$$(((x_0 \oplus x_1) \oplus x_2) \oplus ((e \oplus x_3) \oplus x_4)) \quad \oplus \quad (((e \oplus x_5) \oplus x_6) \oplus x_7) \tag{3}$$

$$((x_0 \oplus ((e \oplus x_1) \oplus x_2)) \oplus ((e \oplus x_3) \oplus x_4)) \quad \oplus \quad (((e \oplus x_5) \oplus (e \oplus x_6)) \oplus x_7) \tag{4}$$

In summary, reducers exhibit several attractive properties:

- Multiple strands can access a reducer without races.
- Reducers are shared without the need for mutual exclusion locks.
- Reducers can be used without significantly restructuring existing code.
- Defined and used correctly, reducers retain serial semantics. The result of a Cilk program that uses reducers is the same as the serial version, and the result does not depend on the number of processors or how the work is scheduled.
- Reducers are implemented efficiently, incurring little or no runtime overhead for synchronization. The cost of accessing a reducer is a hash-table lookup.

**Implementing your own reducer**

Cilk comes with a library of predefined reducers which support many commonly used monoids.

Here's an example usage of the + reducer:

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

// ...
  CILK_C_REDUCER_OPADD(r, double, 0);
  CILK_C_REGISTER_REDUCER(r);
  cilk_for(int i = 0; i != n; ++i) {
    REDUCER_VIEW(r) += A[i];
  }
  printf("The sum of the elements of A is %f\n", REDUCER_VIEW(r));
  CILK_C_UNREGISTER_REDUCER(r);
```

Since the list of boards `queens()` generates is not ordered, we can use a reducer to assemble it.

**Write-up 7:** How can a `BoardListReducer` help you to avoid creating the temporary lists in Section 3.2? Define the monoid: What type of objects will the reducer operate on? What is the associative binary operator? What is the identity object?

Unfortunately, Cilk did not come with a `BoardListReducer`, so let's implement our own in 6 easy steps:

1. Include the Cilk reducer header in your code:

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

2. Define the behavior of the reducer in 3 functions:

```
// Evaluates *left = *left OPERATOR *right.
void board_list_reduce(void* key, void* left, void* right) { ... }

// Sets *value to the the identity value.
void board_list_identity(void* key, void* value) { ... }

// Destroys any dynamically allocated memory. Hint: delete_nodes.
void board_list_destroy(void* key, void* value) { ... }
```

   Remember to cast the `void*` arguments to `BoardList*` before using them. You don't have to worry about the `key` argument (although you might see why it's there, since we previously mentioned a hash-table lookup).

3. Instantiate the reducer type:

```
25 typedef CILK_C_DECLARE_REDUCER(BoardList) BoardListReducer;
```

4. Initialize the reducer for use (it's easiest to put it in global scope):

```
26 BoardListReducer X = CILK_C_INIT_REDUCER(BoardList,              // type
27   board_list_reduce, board_list_identity, board_list_destroy,   // functions
28   (BoardList) { .head = NULL, .tail = NULL, .size = 0 });       // initial value
```

5. You must register and unregister the reducer before and after use. After the declaration of the reducer, but before the first use, insert `CILK_C_REGISTER_REDUCER(X)` to register. When the reducer is no longer needed, insert `CILK_C_UNREGISTER_REDUCER(X)` to unregister.
6. Use `REDUCER_VIEW(X)` to access the value. In this case, it will return a `BoardList`.
7. Once all parallel execution has completed, use `X.value` to query the final value of the reducer.

The full details are at:

`http://software.intel.com/en-us/articles/using-an-intel-cilk-plus-reducer-in-c`

You can find a good tutorial with a working example at:

`https://www.cilkplus.org/docs/doxygen/include-dir/page_reducers_in_c.html`

**Write-up 8:** Use a reducer to parallelize `queens`. Record the parallel execution time using `awsrun8 ./queens`. Verify that the answers you're getting are consistent with the serial code from before. **Note: Cilksan may report races if you do not explicitly call `__cilksan_disable_checking()` and `__cilksan_enable_checking()` before and after accesses to the reducer object.** How does the parallel code with reducers perform compared to the parallel code without?