Building a parallelism profiler for Cilk computations

In this project, you will implement a simple serial tool for Cilk programs — a parallelism profiler for Cilk computations. That is, your profiler will execute a Cilk computation serially, instrument spawn and sync statements during execution, and compute work and span of the computation based on the instrumentation.

You are allowed to work in a team of two for this project. The project is due before midnight, Friday 03/27. You should commit your code before midnight and commit a PDF file of your writeup in your code repository.

Compilation of a Cilk Plus program

Since the tool needs to instrument spawn and sync statements during execution, the user application needs to be compiled to include these instrumentations. To understand how the code is instrumented, we shall first examine how a Cilk Plus program is compiled.

In Cilk Plus, a spawn statement is compiled into a sequence of instructions including multiple calls into the runtime. Likewise, a sync statement is compiled into a sequence of instructions with calls to the runtime. In order for you to implement the tool, you need to understand how spawn and sync are compiled. The compilation of Cilk Plus code is described in details in the document included with your code base: reference/cilk_plus_abi.pdf [2]. We will briefly summarize the compilation here, but the ABI document servers as a good reference on how the compilation is done.

In Cilk Plus, every spawn statement is extraced into a separate function called a spawn helper function. The spawn helper is a closure which includes all information necessary (including capturing reference to local variables within the Cilk function that spawnsws) in order to execute the spawn statement. In particular, a spawn statement that looks like the following:

\[
x = \text{cilk\_spawn}\ f(y);
\]

becomes a sequence of statements shown in Figure 1.
void spawn_f(int *x, int y)
{
   __cilkrts_stack_frame sf;
   __cilkrts_enter_frame_fast(&sf);
   __cilkrts_detach();
   *x = f(y);
   __cilkrts_pop_frame(&sf);
   if (sf->flags)
      __cilkrts_leave_frame(&sf);
}

...  
if (!setjmp(frame.ctx))
   spawn_f(&x, y);
...

Figure 1: Compilation of the spawn statement x = cilk.spawn f(y), where x and y are both int and x is a local variable of the spawner.

Note that the setjmp allows the Cilk Plus runtime to capture the stack state of the Cilk function that spawns, thereby allowing a thief to potentially resumes the function at the continuation of the spawn statement.

In Cilk Plus, every Cilk function and every spawn helper has its own shadow frame (the frame that gets pushed onto the deque for bookkeeping). The reason why a spawn helper needs its own shadow frame is because, in Cilk Plus, one can spawn a C function, and a C function doesn’t have a shadow frame. Allocating a shadow frame for the spawn helper allows a victim executing a spawned C function to return back to the stolen parent in the event that the continuation of the parent gets stolen.

Thus, in the code of a spawn helper, you can see a declaration of the shadow frame (i.e., sf). The __cilkrts_enter_frame_fast initializes the shadow frame, and the __cilkrts_leave_frame performs the corresponding cleaning up. The __cilkrts_detach performs the push operation, pushing the continuation of the parent frame onto the deque. That is, right after detach, the parent (i.e., the function that spawned) becomes stealable. Once detach occurs, the worker subsequently calls the spawned function. When a worker returns from a spawned function (back to the spawn helper), the helper subsequently invokes __cilkrts_pop_frame and __cilkrts_leave_frame. The __cilkrts_pop_frame performs some cleaning up, but the actual popping the parent off the deque is performed in __cilkrts_leave_frame; this is the point when a worker checks to see if the parent (the

1The ABI document makes a distinction between __cilkrts_enter_frame_fast and __cilkrts_enter_frame. For the purpose of this project, you can ignore the distinction.
function that spawned) has been stolen. If so, the pop operation causes the worker to return back to runtime to check if the parent stolen can be resumed, and if not, perform work steal. For the purpose of this tool, \_cilkrts\_leave\_frame will never cause the worker to return back to runtime, since the tool executes a Cilk computation serially, and thus a function can never be stolen.

Similarly, every sync statement is compiled into the sequence shown in Figure 2. The \_cilkrts\_sync is a heavy-weight runtime call to perform a sync, which potentially suspends the frame if the function is not ready to sync. Again, for the purpose of this tool, \_cilkrts\_sync should never suspend (in fact, it should never get called because the flags CILK\_FRAME\_UNSYNCHED should never be set), since the tool executes Cilk computations serially.

```c
if (frame.flags & CILK_FRAME_UNSYNCHED)
{
    if (!__builtin_setjmp(frame.ctx))
        __cilkrts_sync(&frame);
    /* Function is now synched. An asynchronous exception may be pending. */
}
```

Figure 2: Compilation of a sync statement.

Finally, since every Cilk function has its own shadow frame, at the beginning of a Cilk function (at least before it executes its first spawn statement), a shadow frame is declared and \_cilkrts\_enter\_frame\_fast is invoked. Similarly, \_cilkrts\_leave\_frame is invoked before returning. Also note that, the compiler inserts a sync statement before returning from a Cilk function if one is not already present.

Instrumentation of a Cilk Plus program

The Lopata machines have been installed with a special Cilk Plus compiler that will compile instrumentations into a Cilk Plus program. These instrumentations allow one to build a tool that analyze various things about a Cilk Plus program. Specifically, the following instrumentations have been inserted into the compiled code:

- cilk\_enter\_begin
- cilk\_enter\_helper\_begin
- cilk\_enter\_end
- cilk\_spawn\_prepare
- cilk\_spawn\_or\_continue
- cilk\_detach\_begin
• cilk_detach_end
• cilk_sync_begin
• cilk_sync_end
• cilk_leave_begin
• cilk_leave_end

The name should suggest what it does — for instance, cilk_enter_begin is invoked before a call to __cilkrts_enter_frame_fast in a Cilk function, whereas cilk_enter_helper_begin is invoked before a call to __cilkrts_enter_frame_fast in a spawn helper. cilk_enter_end is invoked after a call to __cilkrts_enter_frame_fast in both a Cilk function and a spawn helper. See document reference/LowOverheadAnnotations.pdf [1], section 4.1 for details.

Building your profiler

If you look in the proj3 directory, you will see a file workspan.cpp and its header workspan.h. The Makefile is setup so that it will compile a set of benchmarks (in bench dir) with this special compiler that inserts instrumentations. The workspan.cpp is the main file you should modify to implement the tool. The Makefile is setup to link the benchmark with the tool implementation defined in workspan.cpp.

Besides the instrumentations, workspan.cpp also defines three other functions:
• cilk_tool_ensure_serial_execution
• cilk_tool_start
• cilk_tool_stop

The function cilk_tool_ensure_serial_execution sets up the environment so that the Cilk computation executes serially. This function has been defined for you, and you should not modify it. cilk_tool_start and cilk_tool_stop indicates the beginning and end of the parallel region that your tool should profile. That is, you should ignore any spawn and sync statements outside of calls to these functions. (See boolean flag timing in the code.)

For the purpose of this project, you can assume that for any benchmark that we will use to test your tool, cilk_tool_ensure_serial_execution is invoked at the very beginning of the main function, and cilk_tool_start and cilk_tool_stop are invoked exactly once respectively (i.e., we will measure only a single parallel region), and that the parallel region the tool measures is well-nested (i.e., all spawned functions have returned when cilk_tool_stop is called).

Finally, the proj3 directory also contains a testing script text.sh that invokes each benchmark three times. You can see a reference timing output in test.out. As long as your tool output timing within 10% range of the sample output, you will get full credit.
void A1 () {
    B();
    cilk_spawn C();
    D();
    cilk_sync;
    E();
}

void A2 () {
    B();
    cilk_spawn C1();
    D1();
    cilk_spawn C2();
    D2();
    cilk_sync;
    E();
}

(a) (b)

**Figure 3**: Cilk pseudocode for two simple simple Cilk functions. (a) A simple Cilk function, which we use to illustrate the problem of computing span. (b) A more elaborate Cilk function, which we use to illustrate the more elaborate span computation that Cilkprof performs.

We realize that the timing can sometimes be off, such as shown for matmul in test.out.\(^2\) During testing, we will run your tool multiple times and go by the runtime of the “majority run” (such as parallelism of 3200 in the case of matmul).

**Computing work and span**

To compute the work and span, we will use an algorithm called “Cilkprof,” described in Cilkprof [3].\(^3\) You will use execution time as a measure for this project, but for simplicity, we shall assume that the Cilkprof algorithm computes the work and span of a function instantiation by counting instructions when we describe the algorithm. (The workspan.cpp include timing functions you can use for time measurement; see comments for how to use them.)

As a concrete example, consider the code shown in Figure 3(a). The span of the function A1 is the sum of the spans of B, E, and the larger of C and D, since C and D operate in parallel. Cilkprof computes the span of A1 during a serial execution in which it keeps track of what subcomputations are logically in parallel. Initially, it sets the span of A1 to 0. After it executes B, the algorithm adds the span of B, which is computed recursively, to the span of A1. After it executes C, however, because C is spawned, the Cilkprof algorithm must determine whether C or its continuation, D, has the larger span. To make this determination, the Cilkprof algorithm stores the spans of C and D separately until it executes the cilk_sync. At that point, Cilkprof chooses the larger of the spans of C and D and adds it to the span of A1, producing the final result.

More generally, to compute the work and span of each function instantiation as the

---

\(^2\) This can happen if say, some other process perturbs the running of the Cilk computation on the measured span.

\(^3\) Actually, the Cilkprof algorithm does more than just measuring the overall work and span of a parallel region, but we shall describe only the relevant bit here and refer interested readers to the final paper.
### G spawns or calls F:

1. \( F. w \leftarrow 0 \)
2. \( F. p \leftarrow 0 \)
3. \( F. \ell \leftarrow 0 \)
4. \( F. c \leftarrow 0 \)

### Called F returns to G:

1. \( F. p += F. c \)
2. \( G. w += F. w \)
3. \( G. c += F. p \)

### Spawned F returns to G:

1. \( G. w += F. w \)
2. \( F. p += F. c \)
3. \( \text{if } G. c + F. p > G. \ell \)
   4. \( \text{then } G. \ell \leftarrow F. p \)
   5. \( G. p += G. c \)
   6. \( G. c \leftarrow 0 \)

### F syncs:

1. \( \text{if } F. c > F. \ell \)
   2. \( \text{then } F. p += F. c \)
   3. \( \text{else } F. p += F. \ell \)
   4. \( F. c \leftarrow 0 \)
   5. \( F. \ell \leftarrow 0 \)

### F executes an instruction:

1. \( F. w += 1 \)
2. \( F. c += 1 \)

---

**Figure 4**: Pseudocode for the Cilkprof algorithm to compute the work and span of a Cilk computation. For didactic simplicity, this Cilkprof pseudocode counts instructions in a Cilk computation by incrementing the work and continuation variables at each instruction.

Cilk computation executes, the Cilkprof algorithm maintains four variables for each function frame \( F \) on the call stack. The **work** variable \( F. w \) corresponds to the execution of \( F \) so far. The remaining three variables maintain two possibilities for the critical (i.e., longest) path through \( F \). Both of these paths start at the first instruction of \( F \) and diverge at some **cilk_spawn** instruction \( u \) in \( F \). Each frame \( F \) maintains these two paths using three **span variables**:

- The **prefix** \( F. p \) corresponds to the critical path through \( F \) from the first instruction of \( F \) through \( u \). The path associated with the prefix variable is guaranteed to be on the critical path through \( F \).
- The **longest-child** \( F. \ell \) corresponds to the critical path through the child of \( F \) spawned at \( u \).
- The **continuation** \( F. c \) corresponds to the critical path from the continuation of \( u \) through the last instruction executed in \( F \).

Figure 4 gives the pseudocode for the Cilkprof algorithm for work and span. Cilkprof operates as follows on a function invocation \( F \). On entering \( F \), all four variables are initialized to 0. When an instruction executes in \( F \), Cilkprof updates the work and continuation variables of \( F \). When a called child function \( F' \) returns to \( F \), the work and span of \( F' \) are added to the work and continuation variables, respectively, of \( F \). When a spawned child function \( F' \) returns to \( F \), Cilkprof decides whether the span of \( F' \) is larger than its previously recorded longest child span, and if so, it replaces its longest child span with the span of \( F' \). Finally, when \( F \) syncs, the two paths represented by the longest child and
by the continuation join, and Cilkprof chooses the longer of these two paths to contribute to the span of $F$.

This pseudocode is more complex than our earlier example suggested in order for the Cilkprof algorithm to handle more complex cases, such as is shown in Figure 3(b). In this example, suppose that the true span of $A_2$ is the sum of the spans of $B$, $D_1$, $C_2$, and $E$. After the Cilkprof algorithm executes $D_1$, it stores each of the spans of $B$, $C_1$, and $D_1$ in the prefix, longest-child, and continuation variables for $A_2$'s instantiation, respectively. After Cilkprof executes $C_2$, it checks whether the span of $A_2$ includes the span of $C_1$ or the span of $D_1$ plus that of $C_2$. In this case, it chooses the span of $D_1$ plus that of $C_2$, and updates the span variables such that the prefix is the span of $B$ plus that of $D_1$, the longest-child is the span of $C_2$, and the continuation is 0. After executing $D_2$, the span of $D_2$ is saved in the continuation variable, and the subsequent `cilk_sync` causes Cilkprof to compare the spans of $C_2$ and $D_2$, as in the first example.

Writeup

Describe what challenges you encounter in the writeup, and if you have a partner, what you did and what you think your partner did. Also include with your writeup an output of running the `test.sh` script.

References

