1 Summary

We implemented a parallel version of 15-210 SML sequence library using Poly/ML. This library provides parallelized versions of many useful higher-order functions in a way that is entirely compatible with the existing sequential version of the library.

2 Background

In 15-210, students are taught principles of parallel algorithm design, with an emphasis on analysis of the span (or critical depth) of algorithms. In programming assignments, students are assigned to build programs that have certain asymptotic behavior in their span. They do so with the help of a library of higher-order functions, which is another emphasis of the course. In their analysis of the span of their algorithms, they reason from a spec of the library, which assumes efficient parallel implementation of its methods. However, the actual implementation they are given does not meet this spec—it runs purely sequentially. This can be confusing for many students, who see no improvement in their programs’ performance despite proving that the span has been improved.

On the other hand, some parallelism has been implemented in SML. While SML/NJ, the SML interpreter typically used in 15-210, does not support parallelism, there is another (more obscure) interpreter that does: Poly/ML. However, the libraries included with Poly/ML do not expose this parallelism in a way that lends itself well to elegant functional programming; instead, it provides what is essentially just a minimal subset of the Unix pthreads library. We want to provide a library of useful and elegant higher-order functions that run in parallel by using this library, giving SML programmers a simple and elegant way to exploit parallelism in their code.

We these two shortcomings in current SML libraries to solve each other. Poly/ML provides the basic primitives necessary for writing parallel SML code, while the 210 sequence spec gives an excellent comprehensive list of useful higher-order functions with which to interface with Poly/ML’s low-level threading.

3 Approach

3.1 Choice of Poly/ML

We originally wished to use SML/NJ as our interpreter, as it is one of the most widely used SML interpreters, as well as being the one used in 15-210 and other CMU classes that use SML. However, it has no support for any sort of parallelism (it has an extension called Concurrent ML that originally appeared to provide a threading library; however, it turns out that this only supports executing threads interleaved on a single core).

Therefore, we settled on using Poly/ML, an interpreter that has support for executing threads simultaneously on a multi-core system. This was not a huge setback, as SML/NJ and Poly/ML are essentially entirely compatible. However, it did bring some of its own minor issues: among others, the SML/NJ compilation/build management system (used widely to manage libraries in an elegant way) does not exist in Poly/ML. While this was not a huge problem within the scope of our project, it could present an (admittedly quite small) hurdle to further adoption.

3.2 Threading Infrastructure

There are three primary threading constructs provided by Poly/ML: a fork function (but no join function!), mutexes, and condition variables. These do not lend themselves directly to writing elegant code, so we abstracted
them away with two functions: makeThread and waitThread. makeThread takes as input a function (of type unit → 'a) and creates a new thread that will execute the given function (and store its result at a memory location it allocates). It returns a reference to where the thread’s output will be stored, as well as various references and synchronization primitives necessary to later determine when the thread has completed.

waitThread is essentially the same as the pthreads join function; it takes as input the items output by a call to makeThread, waits for the thread to terminate if it hasn’t yet, and returns the thread’s functions output. The use of these two functions makes it fairly straightforward to write simple fork-join style parallel code.

3.3 Implementation of map

With this threading infrastructure beneath it, the map implementation is fairly straightforward. We first divide the input into a number of evenly-sized subsequences proportional to the number of virtual cores on the machine. We then spawn one thread for each core, which iterates over its assigned tasks sequentially. Finally, we join all the threads and recombine the subsequences into a result list to return to the calling function.

3.4 Implementation of Other Functions

We found that most of the other functions in the library could be elegantly expressed in terms of map, as well as a few other functions (particularly reduce, tabulate, filter, and scan) that we implemented using map to express all necessary parallelism.

This made it very easy to add more functions to the library after the first few were written, without needing to a few express parallelism beyond that done by other, already-implemented, higher-order functions. In fact, several functions were effectively parallelized simply by taking the existing sequential version and having it use some of our parallel functions rather than the equivalent sequential ones – a process requiring no code changes.

The streamlined process of writing later functions exemplified the primary function of our interface. With access to a few neater higher-order functions that are intrinsically parallel, writing larger programs on top of this becomes much more efficient. Instead of worrying about when to join our threads (and in the case of Poly/ML, how to join them), we are free to focus on how to best apply this parallelism efficiently and cleanly.

4 Results

It would not be practical to provide individual performance analysis of every function contained in our library; instead, we will focus on a few of the more commonly used ones, upon which most other functions/user programs are based: map, reduce, and filter. For testing purposes, the function passed to each higher-order function did nothing but (slowly/naively) calculate the 40th Fibonacci number. This simulates a scenario in which the task that we either perform on each list element or to combine two elements is slow, but runs in constant time and is almost entirely CPU-bound (other than memory overhead from recursion). While not the most realistic workload, it provides the most transparent insight into the performance of our library itself.

These three functions’ performance when run on a six-core hyperthreaded machine is illustrated in figures 1, 2, and 3 respectively. When run with significantly more tasks than cores, all three of these functions produce a speedup of approximately 7.5x over their corresponding sequential implementations. This is near-optimal for these tests: while there are 12 virtual cores/execution contexts on the machine, there are only 6 real cores. Because our tasks are mostly CPU-bound, they cannot derive significant benefit from hyperthreading; as such, it is unrealistic to expect a speedup that is very much higher than 6x. The only reason that that is obtained here is probably due to
4 RESULTS

Andrew Smith — adsmith@andrew.cmu.edu
David Rayson — drayson@andrew.cmu.edu

Figure 1: Performance of map

Figure 2: Performance of reduce
the Fibonacci computation needing to access memory to keep track of recursion, creating a little bit of memory latency that hyperthreading can hide.

5 Conclusion

Our implementation successfully takes advantage of parallelism cleanly in a functional context. It has accomplished its goal of enabling SML programmers to easily take advantage of parallelism in a natural functional way and with familiar functional programming paradigms, especially for students or veterans of 15-210. We hope that it will prove useful as an educational tool, or even as a practical library.

6 References


7 Work Done by Each Student

Equal work was performed by both project members.