Carnegie Mellon 15-418 Final Project: Cilk Fork-Join Parallelism Library

Yonah Goldberg
ygoldber@andrew.cmu.edu

Jack Ellinger
jellinge@andrew.cmu.edu

Project Web Page **GitHub Repository**

Summary

We implemented a fork-join parallelism library that mimics the core functionality of Cilk. Users can create programs that easily fork off tasks to run in parallel. In the background, our schedulers manage thread scheduling. We also created a bench-marking suite of six programs with widely different workloads to test the performance of our schedulers.

Our best scheduler, ChildSchedulerLF, achieves an average 5.8x speedup across our test suite, whereas Open-Cilk averages 7.3x speedup. Both of these measurements are for speedup using 12 threads (our largest number of threads). Without our worst-performing test, N-Queens, our best scheduler achieves an average 6.4x speedup across our test suite, whereas Open-Cilk averages 7.0x speedup

Background

About Cilk

Cilk extends C++ with high level parallelism constructs. It allows programmers to easily write parallel code without low level thread management. Cilk implements fork-join parallelism. Users who want to fork off a function call to run in parallel can annotate their call `cilk_spawn my_function()`. When users want to join threads so that main thread execution can not proceed before all forked threads finish, they can insert a `cilk_sync` statement.

The benefit of fork-join parallelism is that Cilk programs behave almost exactly the same with Cilk primitives inserted as if they were not inserted. Therefore, it is extremely easy to parallelize a program quickly. Cilk programs also easily extend to divide-and-conquer algorithm implementations, where there are many recursive calls that can be forked to run in parallel. The goal of a Cilk programmer is to fork off enough units of work so that all worker threads stay busy. They want enough independent work to allow for good load balancing, but not too much independent work so that high granularity does not incur too much runtime overhead.

Open-Cilk is implemented using a work-queue per-thread and a work-stealing policy. When a thread calls `cilk_spawn my_function()`, it indicates that `my_function` may be run in parallel. There are two main strategies you could use when implementing a scheduler:

1. Child Stealing - the function `my_function` is put onto the thread's work-queue and may be stolen by another thread to run in parallel. The main thread continues executing code beyond the function call.
2. Continuation Stealing - the main thread executes `my_function` and puts a continuation of the rest of the main thread's current function on the work-queue.

Open-Cilk chooses the continuation-stealing approach because it tends to be more efficient. With child-stealing, programs run depth-first, placing all spawned function calls onto work queues. Queues can often blow up in size, leading to worse performance.

In contrast, continuation stealing programs run breadth-first, creating just enough work. The drawback of continuation-stealing is it is drastically more complicated to implement without instrumenting a compiler. We spent a lot of time working on a continuation-stealing implementation using `setjmp` and `longjmp`, and successfully managed to get threads to save execution context and have that context resumed by another thread. Unfortunately, it had too many bugs and we ran out of time implementing it. It works **most of the time** on our `quicksort` benchmark, and you can still view the code at `schedulers/cont_scheduler.hpp`.

Library API

We aimed to mimic the core functionality of Cilk, providing a simple scheduler interface that users interact with:

- `T run(std::function<T()> func, int n) // initialize the scheduler with a size n thread pool and run func with n threads`
- `std::future<T> spawn(std::function<T()> func) // spawn a function to be potentially run in parallel. Returns a future on the returned result.`
- `T sync(std::future<T> fut) // Synchronize by waiting for fut to finish. While waiting, steal work from from other task queues. Return the result of fut.`

Our interface works nicely on a lot of different programs. In the worst case, when there are many dependencies, users have to collect vectors of futures and wait/reduce on their results. Consider the following nice implementation of quicksort:

```
void quicksort(int *begin, int *end) {
  if (end - begin <= 5000) {
    seqQuicksort(begin, end);
    return;
  }

  end--;
  int pivot = *end;
  auto middle =
      std::partition(begin, end, [pivot](int x) { return x < pivot; });
  std::swap(*end, *middle);

  auto x = scheduler->spawn(
      [begin, middle]() { return quicksort(begin, middle); });
  quicksort(++middle, ++end);

  scheduler->sync(std::move(x));
}
```

Notice the call to a sequential version of quicksort for small problem sizes. This is an optimization we had to make for our child-stealing schedulers that we explain in the next section. The main point to make is that, true to Cilk style, our library requires minimal additional code, which is attractive for users who want to quickly parallelize programs.

Benchmark Programs

We now break down our benchmark suite, describing each program and its workload.

- **Fibonacci** computes the nth Fibonacci number using the inefficient exponential algorithm. It has many dependencies because to compute the nth Fibonacci number you need to make two recursive calls on $n - 1$ and $n - 2$. Because it has so many sub-problems, we set a stop point where fib stops recurring and computes the result sequentially.

- **Quicksort** sorts an array of size n using the classic quicksort algorithm. Quicksort has zero dependencies. All parts of the array can be sorted in parallel. However, it still has a large recursive tree. Because it has too many sub-problems, we set a stop point where quicksort stops recurring and sorts the array sequentially.
- **NQueens** calculates the number of ways to place n queens on an n by n chess board such that no queens attack each other. There is a lot of sequential work on each sub-problem to check if any queens attack each other, so there is high parallelism.
- **Heat** performs heat diffusion over a 2D grid. It uses Jacobi-type iteration to update the temperature values at each grid point based on neighboring values. Iterations have to be performed sequentially, but there is high parallelism in each iteration.
- **NBody** simulates the NBody problem from lecture where particles move across space with an initial velocity and influenced by gravitational pull. We parallelize across updating the velocities of the bodies. Then we synchronize and then parallelize across updating the positions of the bodies. There's a lot of parallelism across the bodies, but also a good amount of synchronization on each iteration.
- **PFor** computes a parallel for-loop. On each iteration, it spawns off a function to sort a large array. There are no dependencies and high parallelism. This test is meant to look at the difference between continuation stealing and child-stealing. The child-stealer runs inefficiently, immediately putting tasks for each loop iteration on the queue.

Approach

Scheduler Implementations

We now describe our four scheduler implementations in depth, including optimizations and how we arrived at our final implementations.

NoSpawnScheduler

The NoSpawnScheduler is a sequential scheduler that we use when calculating speedup. The following is its API implementation:

- `T run(std::function<T()> func, int n)` - run `func` sequentially.
- `std::future<T> spawn(std::function<T()> func)` - run `func` sequentially.
- `T sync(std::future<T> fut)` - no-op.

SimpleScheduler

The SimpleScheduler is our simplest parallel scheduler. The following is its API implementation:

- `T run(std::function<T()> func, int n)` - set `n` to be the max number of threads that can ever be running at the same time. Run `func`.
- `std::future<T> spawn(std::function<T()> func)` - check to see if there are less than `n` threads running in concurrently right now. If there are less, spawn a thread to run `func` in parallel. Otherwise, run `func` sequentially.
- `T sync(std::future<T> fut)` - block on `fut` until it is ready and return the result.

ChildScheduler

The ChildScheduler is our first complex scheduler. The following is its API:

- `T run(std::function<T()> func, int n)` - Create `n` work-queues, one for each thread. Each queue is a deque, allowing you to push and pop from both sides. Each queue can hold many tasks. A task is a wrapper around a `std::packaged_task<T()>`. Essentially, this is just a function where you can

get a handle on a future to its returned value. The future's result is only ready after the packaged task is run. Each queue is guarded by a unique mutex lock for synchroniztaion.

We place a packaged task for the given function on a queue. We then spawn $n - 1$ threads and have all n threads (including the main thread) enter a workerThread loop. These threads only leave the loop when there are no tasks in any queues **and** no workers are currently working on any tasks. We maintain two atomic counters for this. When all threads are finished, it means all work is done, so the main thread joins all the spawned threads and then retrieves and returns the result from the original packaged task placed on a queue.

The workerThread loop is simple. Each thread repeatedly tries to find work in its own queue. If it does, it runs the task. Otherwise, it steals work from a random task queue to run. An important implementation detail is that workers must take work off their own queue from the same side of the queue that work is pushed. They also must steal work off the opposite side of other worker queues.

The point is that you threads to often resolve their own immediate dependencies. On large problems with large dependency trees, we often run into a problem with the depth-first nature of child schedulers where work queues and call-stacks blow up in size and cause segmentation faults. Taking work off the front of your queue, where the most recent dependencies that need to get resolved are stored, is vital.

- `std::future<T> spawn(std::function<T()> func)` - simply push a task to run `func` onto this thread's work queue.
- `T sync(std::future<T> fut)` - While is not ready, take a task from some work queue and run it. As explained, we always try and steal from this thread's work queue first, on the same side that work is pushed to. If our work queue is empty, steal from the opposite side of a random work queue.

ChildSchedulerLF

The lock free child scheduler has the exact same API implementation as the normal child scheduler, with the exception that instead of using a `std::deque` for work-queues, we use a custom lock-free queue. For the lock-free queue, we adapted code written by Stefan Reinalter. The main idea is that we use a fixed size queue with both a head and tail pointer, functioning like a dequeue. Each thread has one of these queues, and they push and pop at the same side of the queue (call this the head). If there is no work in your current queue, you look at a random queue and attempt to steal work from them. To do this, you attempt to take from the tail end of the queue in order to reduce contention between the thread that owns this queue and the one trying to steal from it. In order to do this with proper synchronization, the head and tail pointers are atomic integers that we only update when we know our change worked. To know if are able to steal/pop a particular value, you need to do a compare exchange that attempts to update the head/tail pointer, and if you succeed, you can then take the value without any race conditions. If you fail, then you just return a null value indicating that you were unsuccessful and the loop that tries to get work will run again (assuming there is still work in the system to do).

Target Machine

We decided to test our schedulers on an M2 Macbook Pro with 16GB memory. We went with this approach because, for one, it was easier during development to test locally. Secondly, the CPU has 12 cores, so it has sufficient ability to demonstrate parallel speedup. We ensured while testing that no expensive processes were running concurrently.

Results

We now describe our results for benchmarking each of our schedulers compared to the performance of OpenCilk on the same program. We tried to pick large enough problem sizes so that running the problem in parallel would be useful and have decent speedup.

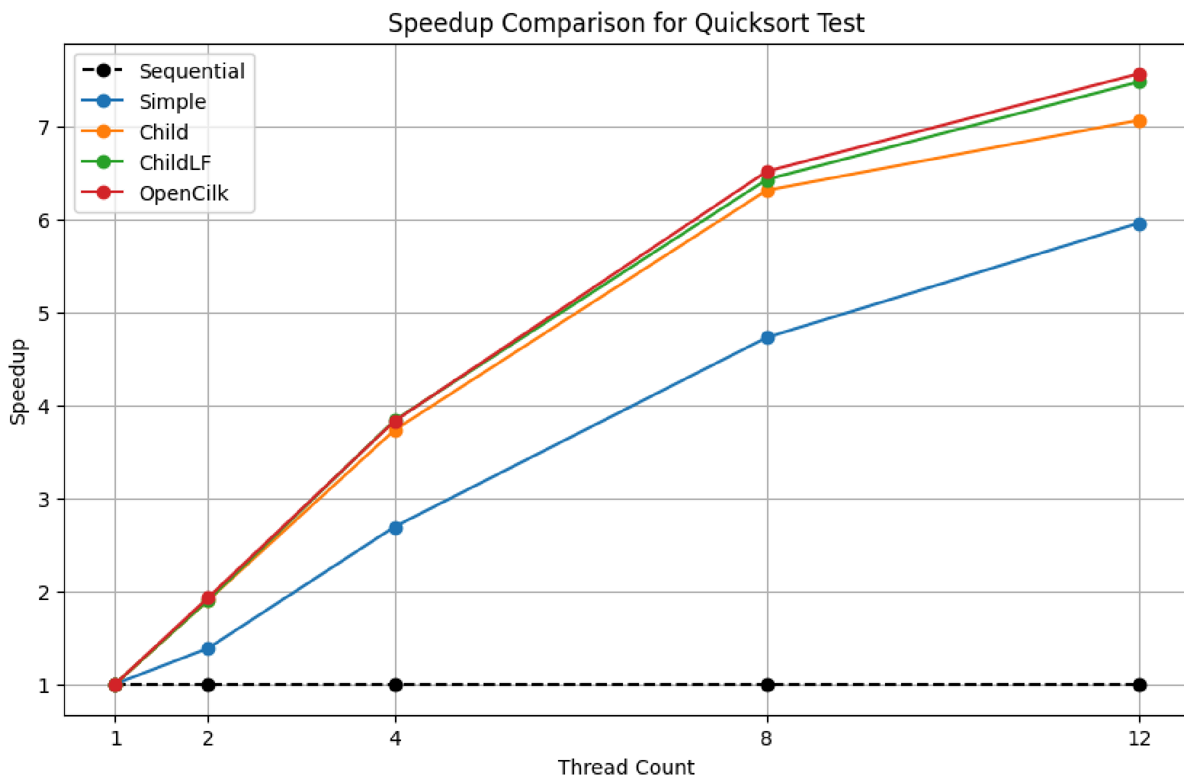
Our benchmark suite essentially consists of only divide-and-conquer implementations of problems, which are the most natural implementations for fork-join parallelism. It is important to note that as problem-size increases, the divide-and-conquer algorithms have larger sequential base cases (sometimes this means manually adjusting when we revert to a sequential solution, as in quicksort). With larger sequential base cases, algorithms spend a larger fraction of the time doing parallel work instead of synchronizing, so the speedup increases. Therefore, it was important that we pick a reasonable problem size for each problem and hold it constant. The main point of this project is to compare different schedulers against each other, so we find no problem in doing this.

Each benchmark is averaged over a specified number of iterations. We chose this number to be large enough so that we did not see a lot of variance in our results.

The baseline for each of these speedup graphs is an optimized sequential version that uses the future framework that our child implementations use. As a result, it has some overhead (about 14% on average) vs the normal sequential code without the futures leading to some superlinear speedup for the OpenCilk implementation. The problem sizes were chosen to take somewhat similar amounts of time across the threads. If we had more time, we would have liked to vary the problem sizes and test on different CPU's to see if these results hold.

Quicksort

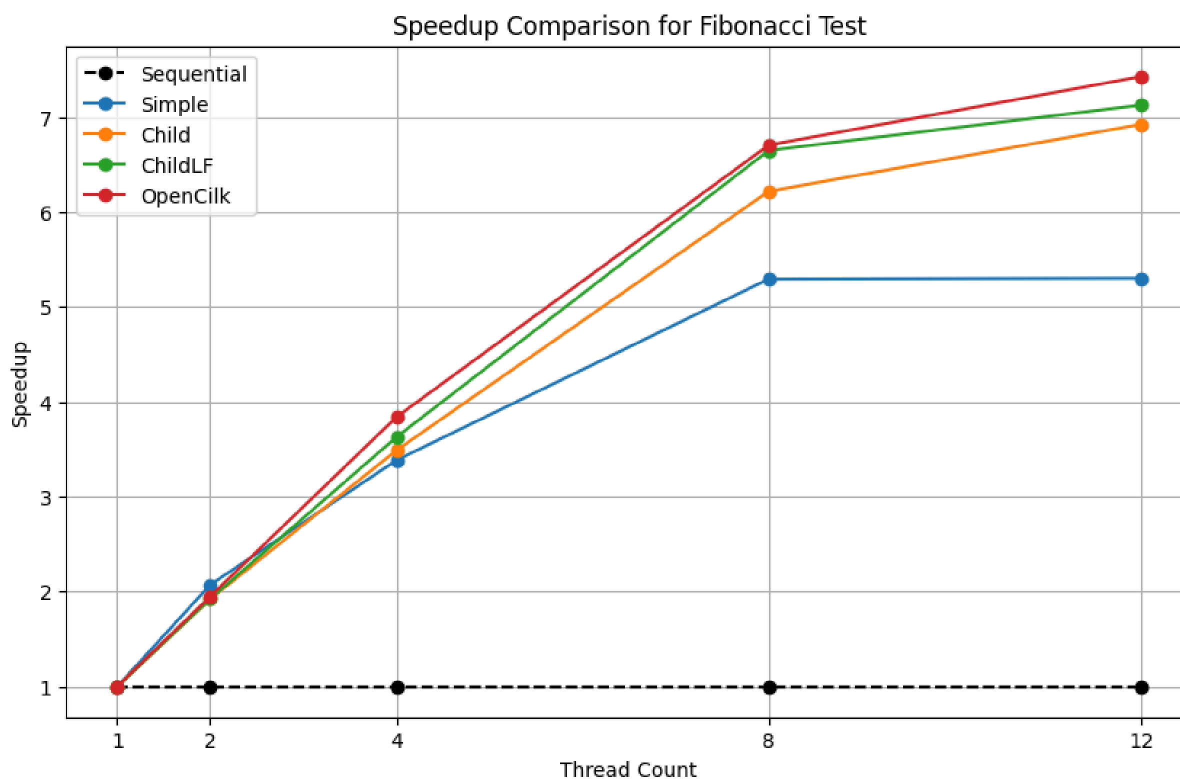
We benchmark quicksort for array size $5,000,000$ and averaging across 10 iterations. The quicksort function defaults to a sequential version when the problem size is less than $5,000$. The following graph shows our results:



Our best scheduler, ChildSchedulerLF, almost matches OpenCilk! The reason for this is that quicksort is one of our easiest benchmarks. There are no dependencies, so threads do not often steal work. The lock free child scheduler outperforms the regular child scheduler because there is low contention for accessing other thread's work queues. As we learned in lecture, lock-free code is best when there is low contention. The SimpleScheduler does not perform as well because of the large overhead of spawning threads. Threads receive uneven sized tasks, causing some to finish early, which makes us repeatedly spawn more threads.

Fibonacci

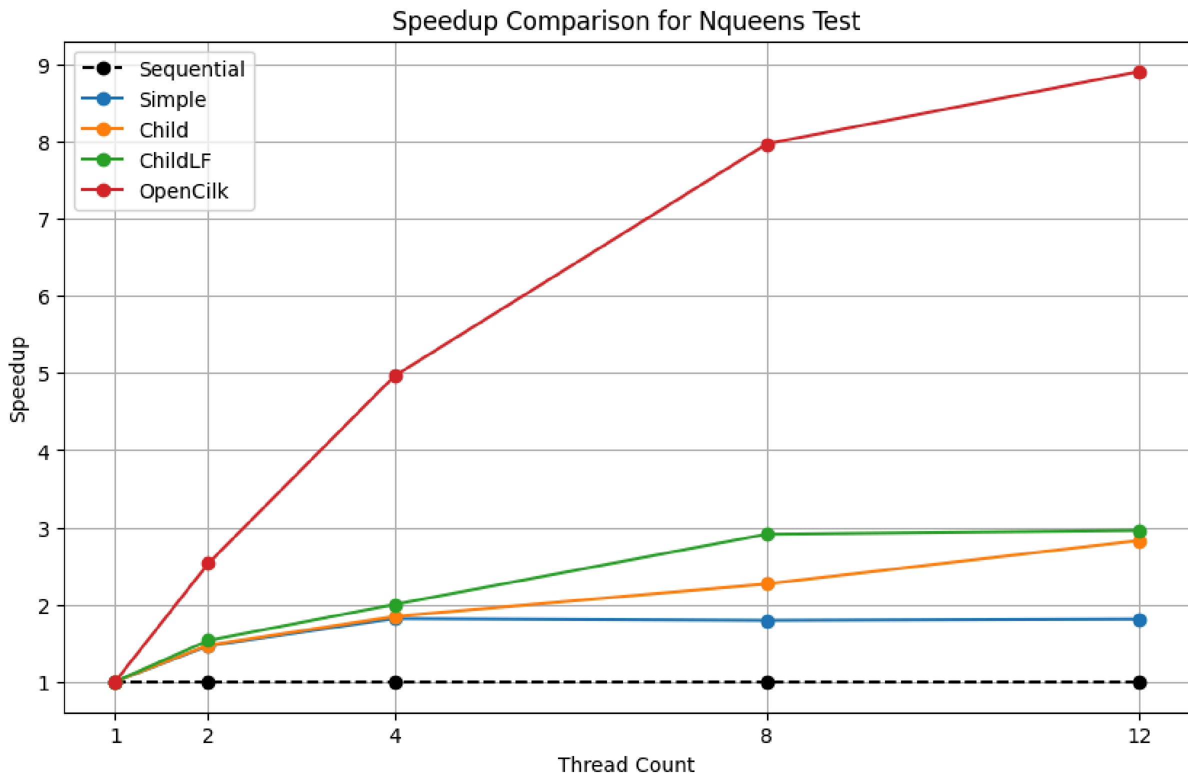
We benchmark fibonacci for n = 45 and average across 10 iterations for each of thread counts. The Fib function defaults to a sequential version when the problem size is less than 20. The following graph shows the speedup results for each of our implementations.



Looking at the results, OpenCilk performs the best followed by ChildLF, Child, and then the Simple Scheduler. Similar to Quicksort, this proved to be a fairly low contention environment where the lock-free implementation of ChildScheduler was able to perform a bit better. Looking at the time spent on compute time vs synchronization, both implementations were mostly doing computation, but the lock-free implementation was able to get the slight edge with reduced time synchronizing as the compare exchange was less costly than a lock. The Simple scheduler performs quite a bit worse as thread count increases, which is likely because the work is not being evenly distributed across the threads leading to a lot of stalling at the end. It also has that extra overhead of spawning the threads and not maintaining a thread pool leading to lesser performance.

N-Queens

We benchmark N-Queens for n = 14 and average across 5 iterations for each thread count. The following are our results:

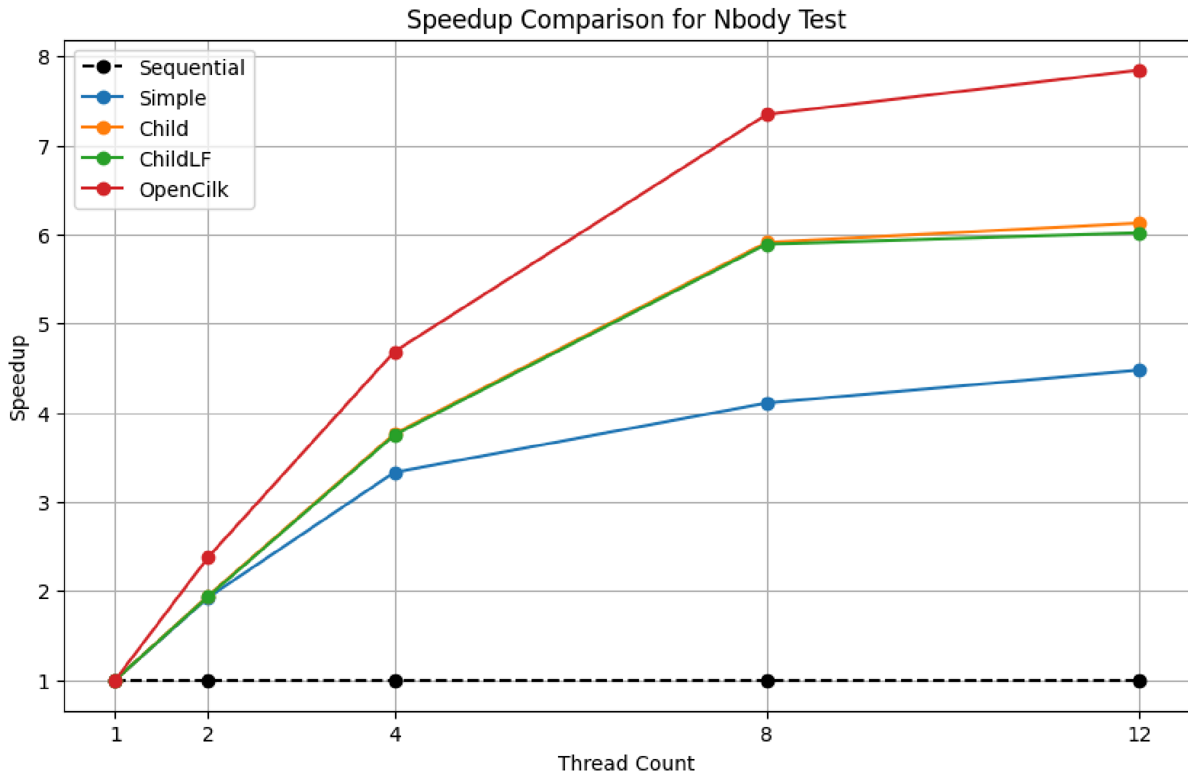


OpenCilk performs drastically better than all of our schedulers. Our simple and child scheduler achieve an unsatisfactory 3x speedup. This test case sometimes gave us problems when benchmarking, so the results might be due to a correctness issue. One thing to note is that clearly there is a good amount of contention because our lock-free scheduler performs poorly, which could have this kind of impact.

Another point to note is this is the only test case where in the end we have a large sum reduction, synchronizing on results of spawned functions. We saw in testing that some threads finish long before other threads, causing a lot of work-stealing and leading to more contention. This imbalance might play a major role in the performance drop-off.

N-body's

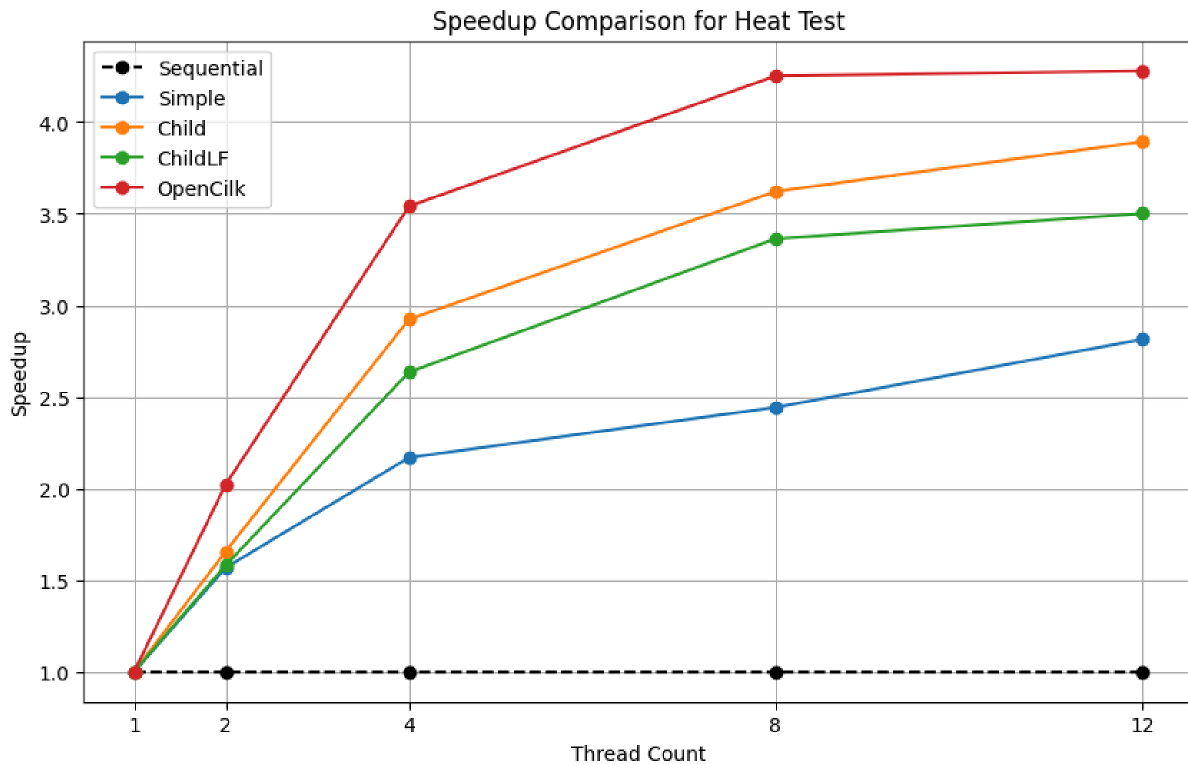
We benchmark n-body's for n = 100000, which means there are 100000 particles in our "universe" that we will be simulating. The following are the results showing the speedup averaged across 10 iterations for each thread count:



The main form of parallelism comes from computing the velocity and position updates for all of the particles. OpenCilk performs the best with our child implementations having almost identical speedup. The Simple scheduler performs a lot worse after that. The continuation stealing approach/compiler optimizations for OpenCilk seem to perform a lot better in higher contention environments, and nbody's requires a lot more synchronization than QuickSort and Fib where we performed a lot better. While lock-free can be faster in a lot of instances, it does a decent amount of failed steals in this test leading to them performing similarly. The simple scheduler, for the same reasons as before, performs worse than our implementations as it does not divide the work properly and has overhead of spawning threads.

Heat Diffusion

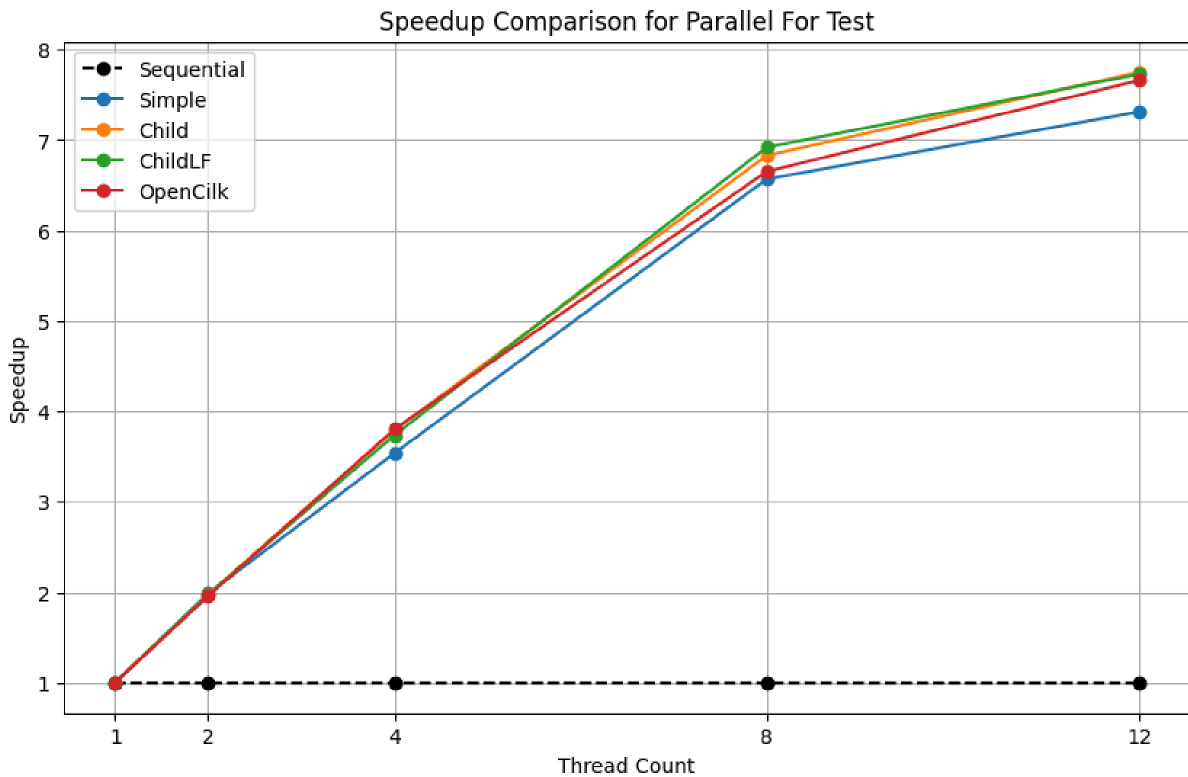
We benchmark heat diffusion 5 iterations using parameters that were hand-tuned by the guy who made this test. The following are the results showing the speedup averaged across 5 iterations for each thread count:



The key insight for this graph is that Child scheduler performs better than lock free scheduler. In this test, failed stealing occurred the most leading to better performance as, when lock free did not successfully steal, it would get a null task back and have to repeat the cycle leading to a major slowdown. Simple scheduler performed worse for the same reasons as before, and OpenCilk was again able to do a decent bit better than our implementation as continuation-stealing seems to perform better in these higher contention environments as continuation stealing will disperse the work better when there are a lot of dependencies.

Parallel-For

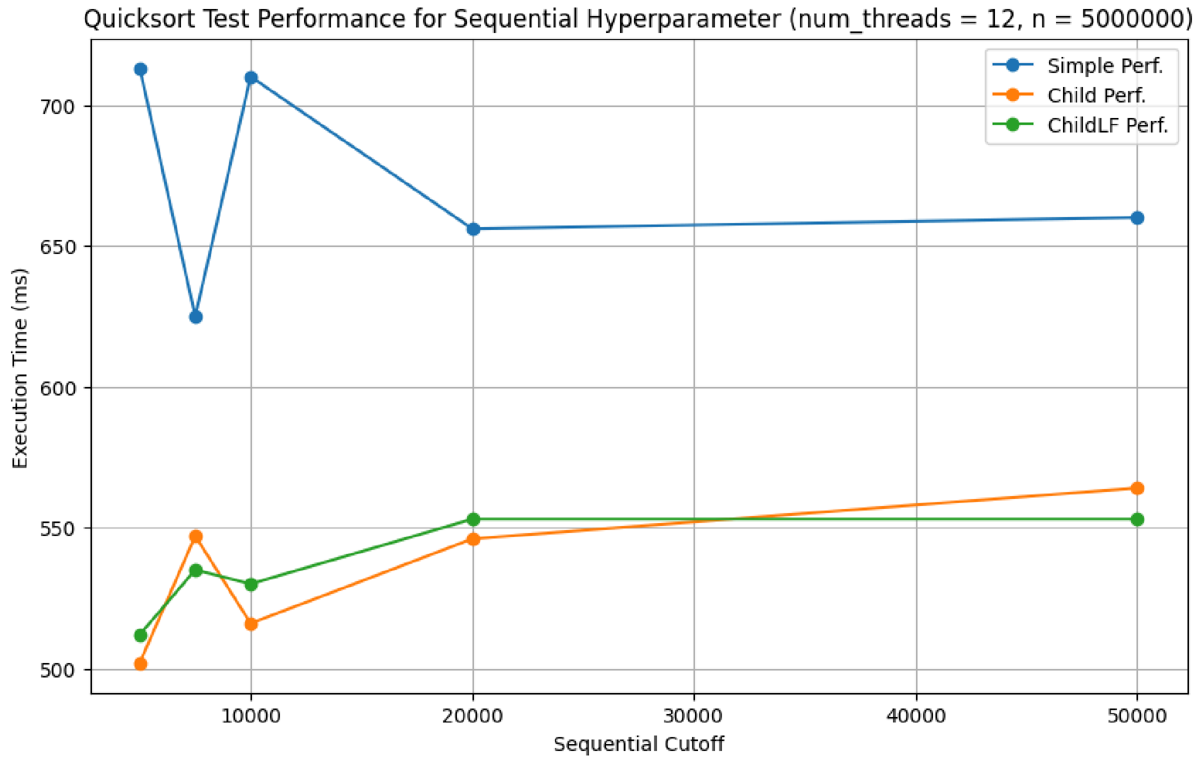
We benchmark parallel-for for 500 iterations of sorting an array of size 100, 000. The following are our results:



We perform well relative to OpenCilk on all of our schedulers. This is an interesting test case because we expected OpenCilk to perform much better relatively. Our child-stealing schedulers will run depth-first and add 500 tasks to the work queues immediately, which we thought would be high overhead. It turns out that the overhead is smaller than we thought. Additionally, parallel-for has little synchronization and even workload distribution. All of these characteristics allow our schedulers to perform well.

Quicksort Test For Sequential Hyperparameter, num_threads = 12, n = 5000000, iterations = 10

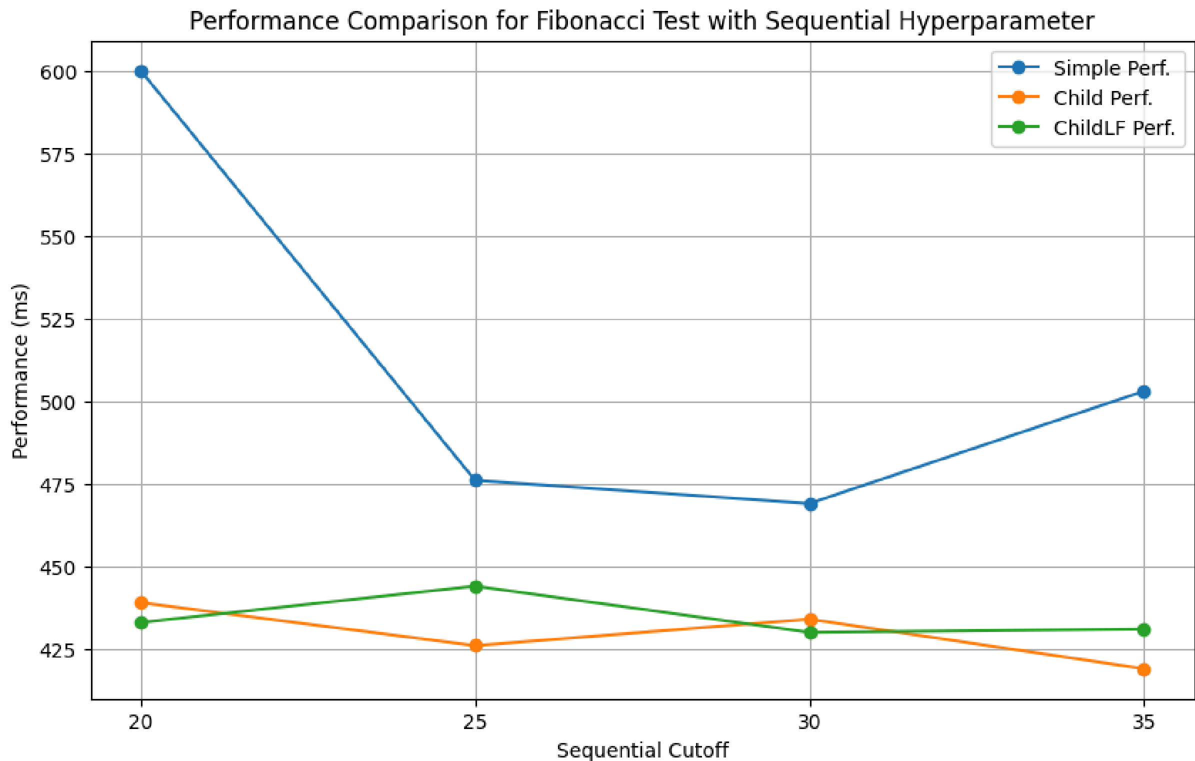
In our implementations, we have a cutoff for when we switch to the sequential version. It is essentially the granularity of the problems we put in our queues



From the graphs, once the problems are large enough, there isn't a huge difference between the speedup as it kind of balances out where the work might be slightly more balanced, but there is more synchronization involved leading to similar performance.

Fib Test For Sequential Hyperparameter, num_threads = 12, n = 45, iterations = 5

In our implementations, we have a cutoff for when we switch to the sequential version. It is essentially the granularity of the problems we put in our queues



Similar to before, from the graphs, once the problems are large enough, there isn't a huge difference between the speedup as it kind of balances out where the work might be slightly more balanced, but there is more synchronization involved leading to similar performance.

Raw Data

Quicksort Test, n = 5000000 iterations = 10

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	3841ms (1.00x)	3841ms (1.00x)	3841ms (1.00x)	3841ms (1.00x)	3841ms (1.00x)
2	3841ms (1.00x)	2773ms (1.3841x)	2020ms (1.9005x)	2018ms (1.9017x)	1990ms (1.93015x)
4	3841ms (1.00x)	1428ms (2.6861x)	1030ms (3.7262x)	1002ms (3.8323x)	1004ms (3.82570x)
8	3841ms (1.00x)	813ms (4.7243x)	609ms (6.3067x)	598ms (6.4207x)	590ms (6.510170x)
12	3841ms (1.00x)	645ms (5.9566x)	544ms (7.0662x)	514ms (7.4799x)	508ms (7.561023x)

Table 1:

Fibonacci Test n = 45, iterations = 5

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	3171ms (1.00x)	3171ms (1.00x)	3171ms (1.00x)	3171ms (1.00x)	3171ms (1.00x)
2	3171ms (1.00x)	1530ms (2.0732x)	1650ms (1.9224x)	1648ms (1.9248x)	1626ms (1.9498x)
4	3171ms (1.00x)	937ms (3.3847x)	909ms (3.4874x)	874ms (3.6260x)	826ms (3.8355x)
8	3171ms (1.00x)	599ms (5.2912x)	510ms (6.2157x)	477ms (6.6403x)	473ms (6.6981x)
12	3171ms (1.00x)	598ms (5.3003x)	458ms (6.9219x)	445ms (7.1258x)	427ms (7.4175x)

Table 2:

Nqueens Test n = 14, iterations = 5

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	15826ms (1.00x)	15826ms (1.00x)	15826ms (1.00x)	15826ms (1.00x)	15826ms (1.00x)
2	15826ms (1.00x)	10810ms (1.4634x)	10748ms (1.4715x)	10319ms (1.5331x)	6251ms (2.5327x)
4	15826ms (1.00x)	8710ms (1.8170x)	8577ms (1.8445x)	7911ms (2.0003x)	3189ms (4.9658x)
8	15826ms (1.00x)	8819ms (1.7940x)	6981ms (2.2666x)	5435ms (2.9127x)	1985ms (7.9756x)
12	15826ms (1.00x)	8743ms (1.8121x)	5593ms (2.8326x)	5344ms (2.9609x)	1776ms (8.9100x)

Table 3:

Nbody Test n = 100000, iterations = 5

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	16339ms (1.00x)	16339ms (1.00x)	16339ms (1.00x)	16339ms (1.00x)	16339ms (1.00x)
2	16339ms (1.00x)	8472ms (1.9273x)	8385ms (1.9476x)	8425ms (1.9390x)	6863ms (2.3816x)
4	16339ms (1.00x)	4903ms (3.3319x)	4339ms (3.7658x)	4356ms (3.7517x)	3486ms (4.6836x)
8	16339ms (1.00x)	3975ms (4.1065x)	2763ms (5.9135x)	2772ms (5.8937x)	2223ms (7.3468x)
12	16339ms (1.00x)	3648ms (4.4759x)	2665ms (6.1332x)	2714ms (6.0180x)	2082ms (7.8395x)

Table 4:

Heat Test iterations = 5

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	16040ms (1.00x)	16040ms (1.00x)	16040ms (1.00x)	16040ms (1.00x)	16040ms (1.00x)
2	16040ms (1.00x)	10232ms (1.5677x)	9681ms (1.6563x)	10145ms (1.5813x)	7933ms (2.02x)
4	16040ms (1.00x)	7396ms (2.1683x)	5487ms (2.9213x)	6087ms (2.6344x)	4531ms (3.54x)
8	16040ms (1.00x)	6567ms (2.4413x)	4430ms (3.6242x)	4770ms (3.3616x)	3773ms (4.25x)
12	16040ms (1.00x)	5701ms (2.8116x)	4121ms (3.8934x)	4583ms (3.5008x)	3749ms (4.28x)

Table 5:

Parallel For Test n = 500, queue_size = 100000 iterations = 5

Thread Count	Sequential Speedup	Simple Speedup	Child Speedup	ChildLF Speedup	OpenCilk Speedup
1	2581ms (1.00x)	2581ms (1.00x)	2581ms (1.00x)	2581ms (1.00x)	2581ms (1.00x)
2	2581ms (1.00x)	1300ms (1.9846x)	1307ms (1.9721x)	1306ms (1.9752x)	1319ms (1.96x)
4	2581ms (1.00x)	730ms (3.5342x)	681ms (3.7871x)	691ms (3.7373x)	679ms (3.80x)
8	2581ms (1.00x)	393ms (6.5638x)	378ms (6.8228x)	373ms (6.9242x)	388ms (6.65x)
12	2581ms (1.00x)	353ms (7.3110x)	333ms (7.7477x)	334ms (7.7186x)	337ms (7.66x)

Table 6:

Quicksort Test For Sequential Hyperparameter, num_threads = 12, n = 5000000, iterations = 10

Sequential Cutoff	Sequential Perf.	Simple Perf.	Child Perf.	ChildLF Perf.
5000	3844ms	713ms	502ms	512ms
7500	3844ms	625ms	547ms	535ms
10000	3844ms	710ms	516ms	530ms
20000	3844ms	656ms	546ms	553ms
50000	3844ms	660ms	564ms	553ms

Table 7:

Fib Test For Sequential Hyperparameter, num_threads = 12, n = 45, iterations = 5

Sequential Cutoff	Sequential Perf.	Simple Perf.	Child Perf.	ChildLF Perf.
20	3127ms	600ms	439ms	433ms
25	3127ms	476ms	426ms	444ms
30	3127ms	469ms	434ms	430ms
35	3127ms	503ms	419ms	431ms

Table 8:

References

- https://blog.molecular-matters.com/2015/08/24/job-system-2-0-lock-free-work-stealing-part-1-basics/
- https://github.com/neboat/cilkbench
- https://www.opencilk.org/doc/

Work Distribution:

- Work done by Yonah: SimpleScheduler, ChildScheduler, half of benchmarks
- Work done by Jack: ChildSchedulerLF, half of benchmarks, other random stuff
- Overall 50/50 work split.