

Parallel Prefix in Pthreads

Algorithm

The goal was to implement a generalized parallel prefix framework using Pthreads. We adopted the algorithm provided in the textbook, figure 5.7 in particular. The code, as mentioned by Dr. Lin in an email, is incorrect. We made a couple of changes in the code. Here is the entire code in Peril-L. The changes that we made are shown in color and bold.

```
int nodeval'[P];
int ltally[P];
forall ( index in (0..P-1) )
{
    int myData[size] = localize( operandArray[] );
    int tally;
    int ptally;
    int stride = 1;

    tally = init();
    for( i=0; i<size; i++)
        tally = accum( tally, myData[i] )

    nodeval'[index] = tally

    while(stride < P)
    {
        if( index% (2*stride) == 0 )
        {
            ltally[index + stride] = nodeval'[index];
            nodeval'[index] = combine( ltally[index + stride],
                                   nodeval'[index + stride]);

            stride *= 2;
        }
        else {
            break;
        }
    }

    // wait for all threads to finish the reduce step before starting the
down-sweep
    // We ran into race conditions in the absence of this barrier
    Barrier.Arrive();
    Barrier.Wait();

    stride = P/2;
}
```

```

if(index == 0)
{
    ptally = nodeval'[P];
    nodeval'[0] = init();
}

// Changed '>' to '>=' here
while( stride >= 1 ) {
    // Introduced the if check here so that the underlying operation
    should happen only if the node is one of the roots at that level
    if( index % (2*stride) == 0)
    {
        ptally = nodeval'[index];
        nodeval'[index] = ptally;
        nodeval'[index + stride] = combine (ptally, ltally[index +
stride]);
    }
    stride /= 2;
}
// Had to add this line for correctness. The program logic guarantees
that nodeval'[index] is full at this point
ptally = nodeval'[index]
for (i=0; i<size; ++i) {
    myResult[i] = scanGen (ptally, myData[i]);
}
}

```

Implementation

The implementation gave us new insights into programming with Pthreads. We implemented useful little primitives like type agnostic Full-Empty variables (FE.h) and Barrier (Barrier.h).

We protect each element of the `ltally` array with a separate mutex. In many cases, like prefix sum computation, the elements of this array are integers only. If we assume atomic reads and writes for integers (where the size of reads and writes is equal to the word size of the machine), then this locking might appear to be an overkill. But we decided to go with it because -

- a) The generic framework can be used for many different applications that may require `ltally` elements to be larger than the machine wordsize. Team Standing application, for example, requires an entire array of as many integers as the number of teams as one element of `ltally`. Reads and writes of these large variables cannot be assumed to be atomic.
- b) The number of elements in the `ltally` array and the number of times they are accessed depends on the number of threads and not on the number of data items. They are quite small and we don't expect a huge performance penalty for protecting them with locks.

The framework is quite generic. For each new application, we just need to specify the four functions – Init(), Accum(), Combine() and ScanGen() along with the type definitions for input, result and tally. Applications are defined in a single header file like this (prefix_sum.h)–

```
// A new class for each application
class PrefixSum
{
public:
    // Each application needs to define 3 datatypes
    typedef int TALLY;
    typedef int INPUT;
    typedef int RESULT;

    // and four functions
    static TALLY Init() {
        return 0;
    }

    static void Accum(TALLY& t, const INPUT& a) {
        t += a;
    }

    static TALLY Combine(const TALLY& l, const TALLY& r) {
        return l+r;
    }

    static RESULT ScanGen(TALLY& p, const INPUT& a) {
        return p+=a;
    }
};
```

In order to run the scan, one needs to call – `Scan<PrefixSum>(inputArray, outputArray, lengthOfArrays, numThreads);`

More complex applications are defined in team_ranking.h and segmented_sum.h

Applications

We have implemented 3 applications.

1. **Prefix Sum.** This is the most basic application for the framework. It generates cumulative sum for a given integer array. The entire code of this application is shown above.
2. **Team Ranking.** This is an application to compute the winning team at any given point in a tournament. The input is an array containing the id of the winning team for each game. For each

game, the output contains the id of the team that has had the highest number of wins till that point. If there are k teams, we keep a k-element integer array as the tally data. The code for this application is present in the included file `team_ranking.h`.

3. **Segmented Sum.** We noticed in the literature ([Scan Primitives for GPU Computing](#)) that a large number of interesting applications like Quicksort require a segmented scan. We implemented this application as a proof of concept for segmented scans. The idea is to have prefix sums of a different sets of data (segments) parallelly in one scan operation. Each element of the input array contains a value and a flag. If the flag is set to true, prefix sum is reset at that element. This application uses the same generic framework used by the previous two applications. `segmented_prefix_sum.h` contains the application code.

Testing

We tested for correctness by first running the code with small input sets (like 16-element arrays). For example, below is a 16-element input data for Segmented Sum application. We processed it with 8 threads, giving two data items to each.

Values:	1	2	3	4	5	6	7	8	9	10	11
	12	13	14	15	16						
Flags:	t	f	f	f	f	t	f	f	t	f	f
	f	f	f	f	f						
Outputs:	1	3	6	10	15	6	13	21	9	19	30
	42	55	69	84	100						

We ran into deadlocks initially. We realized that if some threads move into the second (down-sweep) stage while others are still in the first (reduce) stage, this condition may lead to a deadlock. Our solution was to introduce a barrier between the two stages. A thread can go into the second stage only if all threads have finished the first.

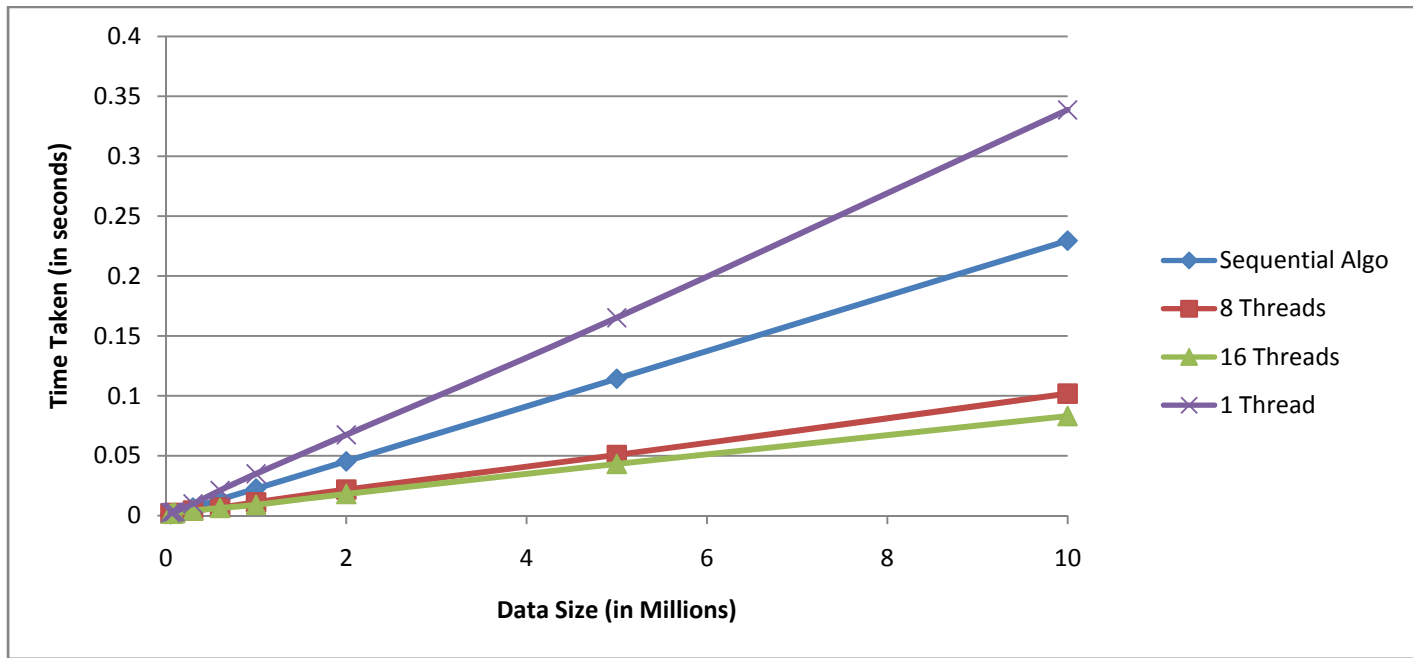
Secondly, we wrote sequential counterparts of our applications (like `team_ranking_sequ.cpp`). Now we could run both parallel and sequential implementations of large amounts of data. By comparing the output generated in the two cases (using a diff tool), we could verify that the parallel program is correct (or has the same bugs as the sequential one 😊)

Performance

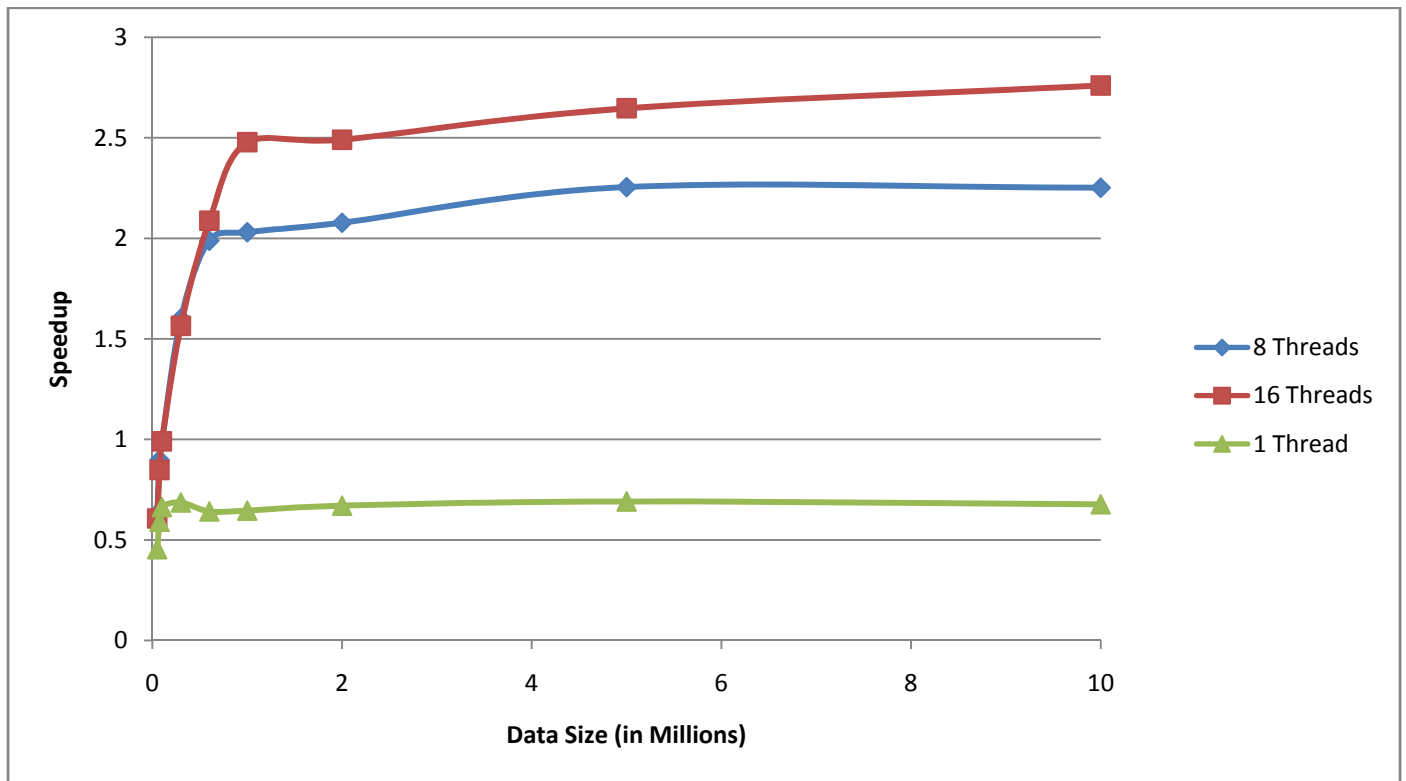
We evaluated the performance of our program on *champion*.

We used the Team Ranking application. We ran our application with 1, 8 and 16 threads with the input data size varying from 50,000 games to 10,000,000 games. We also compared this performance with a straightforward sequential implementation of the algorithm.

Here is the execution time graph –



And here is the speedup –



Speedup at any data size is defined as the time taken by the sequential algorithm divided by the time taken by the parallel algorithm for a given number of threads.

Here is the data from which we generated these graphs. Execution times are in microseconds.

	Sequential Algo	8 Threads	16 Threads	1 Thread
50000	1137	1803	1873	2503
75000	1703	1896	2008	2879
100000	2251	2276	2272	3391
300000	6775	4231	4329	9865
600000	13544	6812	6487	21105
1000000	22600	11129	9120	34979
2000000	45293	21799	18187	67508
5000000	114281	50675	43175	165202
10000000	229490	101910	83148	338714

Conclusions

We can see from the graphs and the table above that

- Sequential algo is always faster than the parallel implementation running with just 1 thread. This is because of synchronization and other overheads in the parallel implementation.
- Parallel implementation running with 8 or 16 threads is actually worse than the sequential implementation for data sizes less than 300,000 elements. The parallelization overheads are justified only for large input sizes.
- It is obvious that N threads do not give N times the speedup. In our case we could at best achieve a speedup of 2.76 with 16 threads.
- Even though *champion* nodes have only 8 cores, we experienced a speed improvement while going from 8 to 16 threads. This seems to be because the OS can smartly schedule 2 threads per core filling any idle cycles that might be there for one thread.