Parallel Prefix Optimization

Benefits from a laddered data distribution

Jason Carver
18.337J/6.338J
Final Project

I tested several different optimizations of parallel prefix on the SiCortex, rejecting a few and confirming the benefits of a novel optimization: the laddered data distribution. I discuss the motivation for optimization, the benchmark framework for testing optimizations, the results of a few standard optimizations, and two novel optimizations and their benefits.

Motivation

The ultimate belief that motivated this project is that parallel programming should be easier. I believe a great way to accomplish this would be to have a cross-compiler that can convert easy-to-write functional code into low-level parallelized code across an arbitrary number of processors. I set out to build a compiler that could convert from a home-brew functional language to C/MPI code. Although I still believe this goal is possible, I had to maintain the project timeline and so was forced to scope down a bit. As part of the early compiler work, I found that a recurring theme was recursive functions that seemed difficult to parallelize at first glance. Over time, I realized that these could often be rewritten with a kind of parallel prefix, as long as each operation applied within the recursive function is associative. As I looked closer at the standard parallel prefix algorithm, it seemed that there were many opportunities for optimization. This is ultimately what my project became about: the optimization of parallel prefix, versus a standard MPI Scan approach.

Benchmark and Framework

In order to properly test my attempted optimizations, I designed a consistent framework in which I could test the changes. The attempted optimizations would be tested against a standard MPI implementation of a cumulative sum. This benchmark implementation had four main steps: a local cumulative sum, pass the total sums across the processors, another local sum, and passing back the final result to the root processor.

Specifically, the tests would run a cumulative sum over 200 billion integers, across 600 processors on the available SiCortex machine. These 200 billion integers are split (nearly) evenly across the 600 processors, and the test framework considers the most important measurement of time to be from the first integer added on processor one to the last integer added on processor 600 and sent back to processor one. This is roughly a start to finish measurement of the whole computation.

Standard Optimizations

The standard optimizations were the natural first step, using things like the compiler -O3 flag, and MPI Scan instead of manually handling message passing and receiving. Even these simple optimizations had some surprising results.

The compiler flag showed a fairly dramatic improvement in speed, giving a roughly 5x improvement over no flag. Part of the compiler flag's effect was to nullify the benefits of a more esoteric optimization: total before incremental sums. The optimization was intended to save time by avoiding the storage call during the incremental sums of the first local cumulative sum. The storage of incremental sums would only be done on the second pass of local cumulative sums. At first, this optimization seemed to be doubling the speed of the first step. Later, this “total before incremental” optimization was tested with the compiler flag on, and the benefit evaporated.

The MPI Scan standard passes data from one processor to the next, using a standard operation to accumulate the data as it goes (the operation being sum in this case). It is not difficult to implement this instead by using MPI Send and Receive calls from one processor to the next. So I tested the difference between MPI Scan and the Send/Receive calls. It turns out that Scan typically gives a 2 percent boost measuring the time for data to pass from the first processor to the last, but it has a significant downside: it is a blocking call. MPI Receive calls are also blocking, but they return as soon as the specified unit of data is received. MPI Scan blocks on every processor until the final unit of data

reaches the final processor. The local processors cannot continue with their second round of local cumulative sums until the MPI call returns. So they get delayed much longer when sending data via MPI Scan, because they have to wait for the data to traverse 600 processors instead of just one. Using this data, I abandoned MPI Scan, and left on the compiler flag for further optimizations.

Communication Optimizations

I attempted two major optimizations on the communication strategy that differed from the standard ones above. I tried a broadcast optimization and a laddered data distribution. These optimizations had to do with the flow of communication and taking advantage of opportunities caused by communication latency.

The broadcast optimization worked by sending results to all relevant processors as soon as they are ready. This should save time because it is much faster to add a few extra numbers than to wait for the cumulative communication delay. For example, when using the broadcast method, the nth processor has to add the results coming in from processors 1 through n-1 (instead of only adding the result from the n-1 processor). However, it only has to wait one length of communication latency, saving n-2 communication latency steps. The downside is that there are many more total communications, adding dramatically to total bandwidth. The total bandwidth used is now n(n-1)/2 instead of n-1, meaning there is a factor of n/2 more bandwidth (in this case 300 times more). Using the SiCortex, which is specified to have a wide bandwidth, this may have been a worthwhile tradeoff. During the tests, it turns out that the broadcast method turned out to be significantly slower. Broadcast was 22.7 percent slower than the standard optimizations above.

The laddering distribution is the main success of this project. It works by taking advantage of the latency in the series of cumulative communication steps required. There is wasted time in the later processors while waiting for the total sum to propagate from the first processor. This wasted time is

because every processor has the same amount of data, and so will finish processing that data at roughly the same time. Using this fact, you can reduce computation time by laddering the data, adding a bit more data on each consecutive processor to fill the gap of wasted time. This strategy leads to a 15.6 percent improvement in start to finish computation time.

A Closer Look at Data Laddering

There is waste in even data distributions. After the second processor finishes calculating the local cumulative sum, it waits for one length of communication latency to receive the result from the first processor. Then the second processor adds the sum to its own and passes it on. The third processor waits after its own calculation for a total of two lengths of communication latency, plus one operation. In most cases, the time cost of one communication latency overwhelms the cost of a single operation (a sum in this case). So the nth processor has to wait for a total of roughly n-1 lengths of communication latency. During this wait, the nth processor does not have any more operations to do, so it sits idle waiting for the communication. This is what I am trying to optimize by laddering the data distribution. If there is a little more data on the second processor than on the first, then it will have something to do while it waits for the result from the first processor. Then there should be even more data on the third processor, because it has to wait even longer.

The amount of extra data added to each processor should be roughly equivalent to the number of calculation operations the processor can do in the amount of a single communication latency. In tests, the ratio of operations to communication is about 25,000. In other words, processor 2 should be able to compute an extra 25,000 operations while waiting for the communication to arrive. Processor 3 should be able to do an extra 50,000 operations while waiting for the communication, and so on. So laddering requires a preprocessing step which calculates how to ladder the data properly. An interesting side effect of this preprocessing step is that it can actually calculate a maximum worthwhile

number of processors to use. If the first processor starts with 25k, and there is an increase of 25k in each processor after that, and there is a total of 150k data, then there is no point in having more than three processors, it would just slow things down because of communication costs. If you request 10 processors when calling this code, it will automatically downgrade your request to only three.

In testing, I tried a variety of settings for the calculation to communication ratio including 6k, 12k, 50k, and 75k. All of them performed worse than the predicted optimal setting of 25k.

The total amount of theoretical time possible with this optimization is related only to the ratio of calculations to communication and the number of processors used. The total number of computations hidden in the otherwise idle time is roughly 25k * p*(p-1)/2 or 4.5 billion computations in 600 processors. These computations do not add to the start to finish time of the original computation, so in a sense come for free. They can be removed from the original block of evenly distributed data, meaning each processor can get 25k * (p-1) / 2 = 7.5 million less data points, reducing the computation time significantly.

Unfortunately, data laddering takes a bit of preparatory work. It requires the programmer to test the ratio of calculation to computation, or at least have a reasonable guess before testing out some reasonable options. Besides that, there is no reason this concept could not be incorporated into a library just as accessible as MPI Scan. The nature of the computation is such that it is not worthwhile for one-time script type calculations (because of the programmer overhead of determining the optimal laddering ratio), but it could be very useful for long-running calculations or computations that need to be repeated often.

So the ultimate suggestion for optimizations of parallel prefix are to stick exclusively with the compiler flag optimization and data laddering. The combination of these two seem to provide the fastest possible implementation of parallel prefix, among the optimizations tried.