Polynomial Evaluation with Parallel-Prefix on Jinx

A Case Study

Patrick Lavin & Harsh Shrivastava

1 INTRODUCTION

The report covers our implementation of a parallel polynomial evaluation algorithm using parallel-prefix.

2 IMPLEMENTATION

This section describes our implementation of the 4 functions required by the project description, Scatter, Broadcast, Parallel_Prefix, and MPI_Poly_Evaluator.

2.1 Scatter

We implemented a naïve version of scatter where the source rank determines what data every other processor needs and then sends it by itself. This takes O(p) steps but sends the minimal amount of data. Since the number of processors does not necessarily divide number of coefficients, the first few processors may have more work to do than the rest. Additionally, none of the other ranks besides the root know how many total points there are, so the root must first tell each process how much space to allocate. This is only a constant overhead though, so we still run in linear time.

2.2 Broadcast

Our broadcast algorithm run in O($log(p)$) time. At the first time step, processor 0 sends to 1. Then processors 0 and 1 send to 2 and 3. This is described in Figure 1. We found this easier to implement than the algorithm provided in class, as it is easier to change the root process on this algorithm. This is accomplished by (1) subtracting the value of the root process from each process's rank (wrapping around mod p if necessary), (2) calculating from whom to send or receive in the regular fashion (as if the root were zero), and (3) adding back the amount we subtracted to the rank(again wrapping if necessary.) The in-class algorithm had trouble with values of p which were not divisible by 2^n. This algorithm doesn't have that problem.

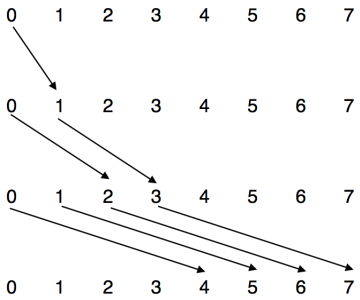

Figure 1: Broadcast Structure

,
. .
DOI:

2.3 Parallel_Prefix

This is the standard Parallel_Prefix (PP) algorithm we learned in class. Each processor is given a list of elements and an operation to perform. It first computes the prefix-op of its own elements. It then follows the typical prefix communication structure to find the total of all elements before it and the total of all elements. After communication, it applies the total of all elements before it to its own array, to get the proper local result. In this implementation, op can be either addition or multiplication, but it would be trivial to extend this to further associative arithmetic operations.

As each rank must do a send and receive at every step, we use non-blocking receives so that we do not need to worry about which rank sends first. Each rank simply does (1) MPI_Irecv, (2) MPI_Send, and (3) MPI_Wait and the problem is solved.

2.4 MPI_Poly_Evaluator

After building the other functions, MPI_Poly_Evaluator is rather trivial. First, a run of Parallel_Prefix using multiplication on the array [1, x, x, ..., x] gives us the required powers of x. Each processor can then evaluate a portion of the polynomial by multiplying the correct coefficients and powers of x. A second Parallel_Prefix sums over this partial result and the final processor ends up with the polynomial value we wanted.

3 EXPERIMENTS

The Jinx cluster has 24 nodes with 2x Xeon X5650 each. Students are only allowed to use 8 nodes at a time, and we had issues with our job crashing when running on 6+ nodes, so we ran our experiments on 4 nodes. In each of our runs, we use the same number of processes-per-node (ppn). We ran tests with ppn of 1-12, as 12 is the number of cores available per node. As we always run on 4 nodes, $p = 4 * ppn$. (After we started writing the report we realized that there are actually 24 hyper-threads available per node, when we thought there were only 12. We mistakenly thought there was only a single CPU package per core.) We run the experiments for 3 different values of n, up to $n = 1.0e8$. Each run, described by (n, ppn), is performed 10 times and each runtime is plotted, as opposed to averaging the times.

We first used random input but found the results hard to verify due to floating point operations not being associative. So we instead set $x = 1$ and $c_i = 2$. While some values of x caused the program to run unrealisticly fast (abs$(x) \leqslant \frac{1}{2}$ causes x^n to converge to 0, specifically), we found that the timings with $x = 1$ were no different from random x values.

To get serial timings for verification and timing comparisons, we used a single batch node. There was not significant variability in the runtimes, so only a single time is plotted. The algorithm computes a power of x at every step (using the previous power), multiplies by the proper coefficient, then sums into an accumulator variable. Thus it runs in O(n) time and requires O(1) additional space.

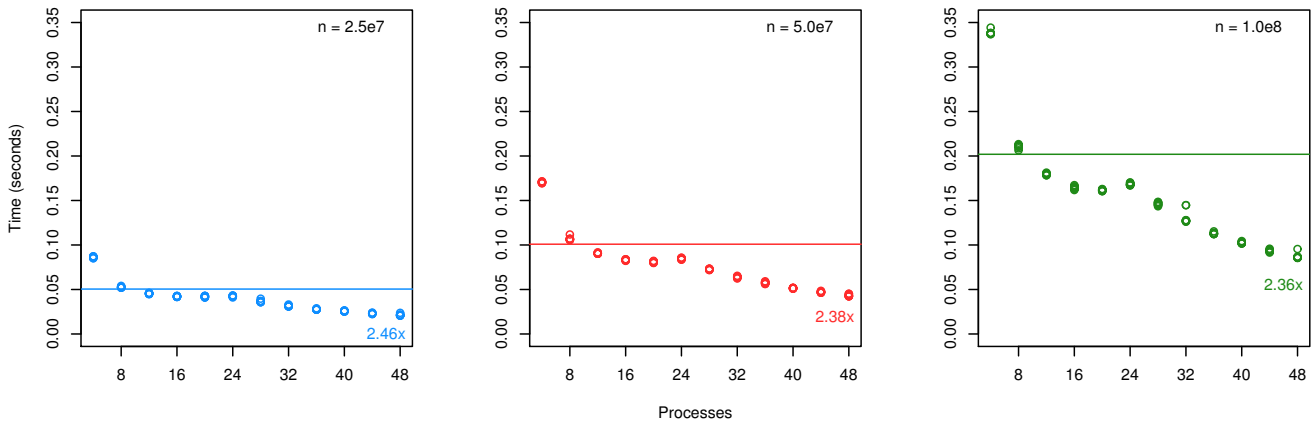

Figure 2: Parallel Runtimes. The degree of the polynomial. is in the upper-right of each plot and the serial runtime is displayed as a horizontal line. The time reported is Broadcast + MPI_Poly_Evaluator.

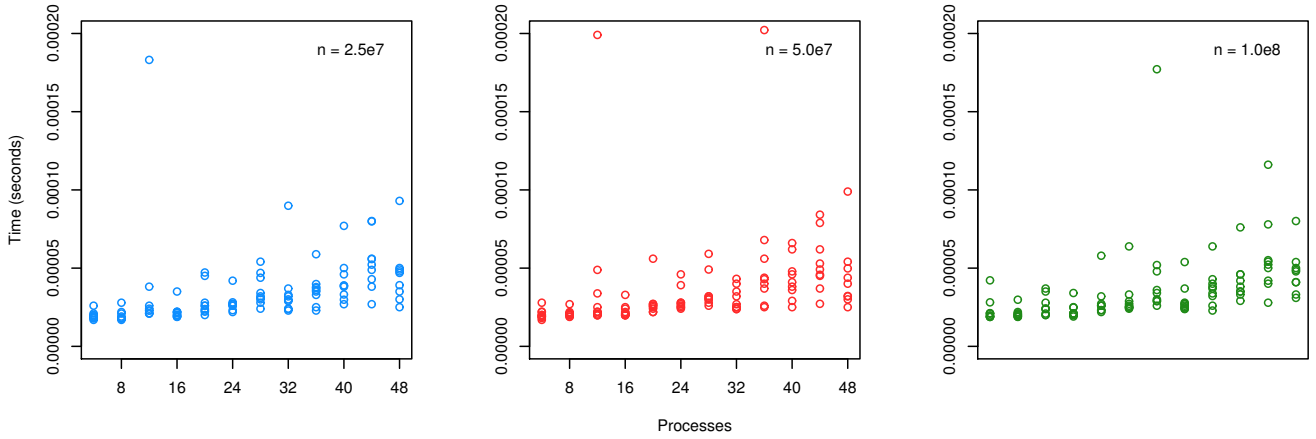

Figure 3: Broadcast runtime.

4 RESULTS

The results for the entire algorithm are displayed in Figure 2. Note that this displays p, not ppn.

Thankfully, we beat the serial implementation! It it interesting to note that this happens at $p = 12$ for every value of n. Sadly, we only get to about 2.3x speedup for each n, meaning we likely have a very inefficient algorithm on our hands.

An interesting point on the graph is at $p = 24$. This is when each node begins to use more than 6 processes, and has do one of two things: (1) communicate between packages, or (2) share time for multiple hyper-threads on a single package.

We also included timings for Broadcast by itself in Figure [**?**]. We have omitted a few outliers, likely caused by network contention. With such a short timescale, we are easily affected by other jobs running on the machine. We note that this grows very slowly, as expected.

5 CONCLUSION

Our parallel algorithm beat the serial version for a large value of p, but it is disappointing that we couldn't beat it sooner. However, since we are confident our implementation doesn't waste any time, it is likely that a separate algorithm is needed if we want to do any better.