Jake Christensen W01023683

CS322 WTR 2014

Assignment 2

It's not always easy to guess what computers are doing. This was certainly the case when working with the complexities of concurrency. My close inspection of the performance of the assigned algorithms with MPI has certainly challenged a lot of the assumptions I had made going into the creation of algorithms. I suspect much of this has to do with the hardware I was using, and more with person using it. I don't believe there is a lot of benefit to running a lot of cores on a system that can't support them.

I'll start with my expectations. My assumptions were first that part two would be the slowest throughout, as it isn't really efficient. It involves a lot more message passing than the other two, and I figured this would make it slow. I assumed with a smaller number of cores, part one would be the fastest. Everything is sending directly to one place while both other methods require every node to touch a piece of data before they report. Finally I assumed that as the number of iterations went up, part three would become the fastest, as it allows every core to be doing work simultaneously.

What I found in my results was much different. Across the board algorithm number one was the winner. Its performance also seemed to be less affected by the number of cores than the ring method. Both method one and three have a significant startup cost during the first loop, but after that seem to run at a fairly stable speed.

Type Number 1		Type Number 2		Type Number 3	
5	4.84E-05	5	0.036467	5	1.62E-05
10	1.00E-04	10	0.042801	10	2.00E-04
15	1.90E-04	15	0.046819	15	6.61E-04
20	2.43E-04	20	0.047986	20	0.001394

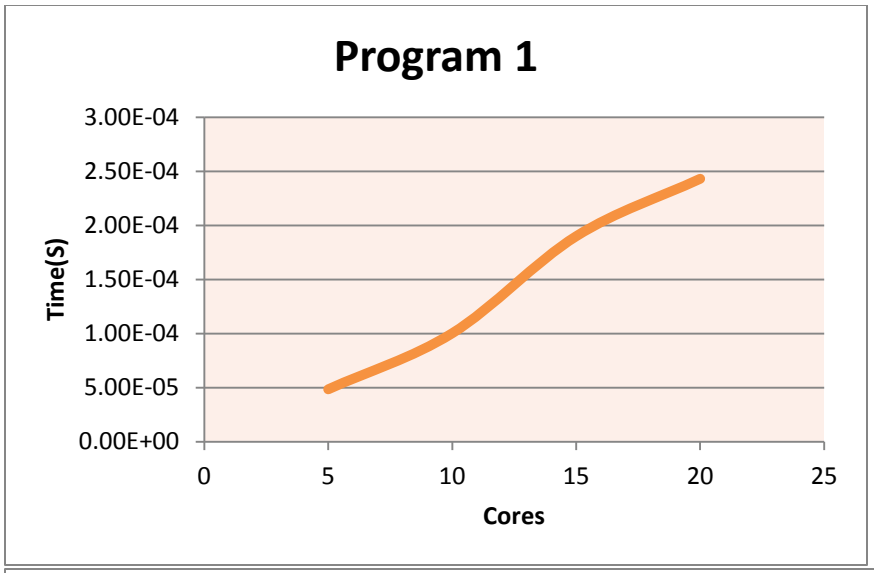
This table shows the number of cores ran along with the average time to finish in seconds.

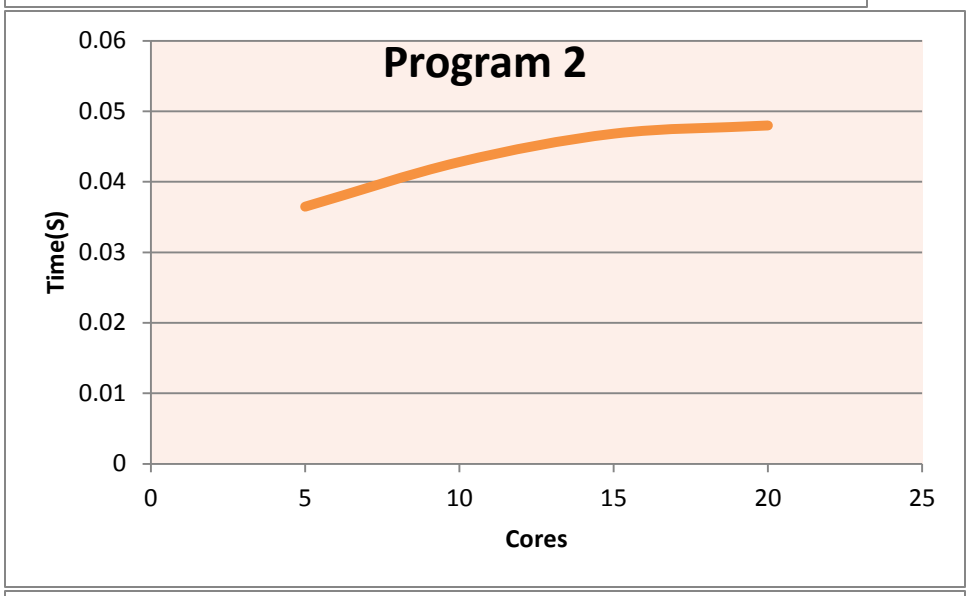
Type Number 1		Type Number 2		Type Number 3	
0	0.001912	1	0.007855	0	0.012405
3	7.77E-05	3	0.00403	3	1.61E-04
15	7.61E-05	15	0.004713	15	1.55E-04
29	7.40E-05	29	0.083755	29	2.44E-04

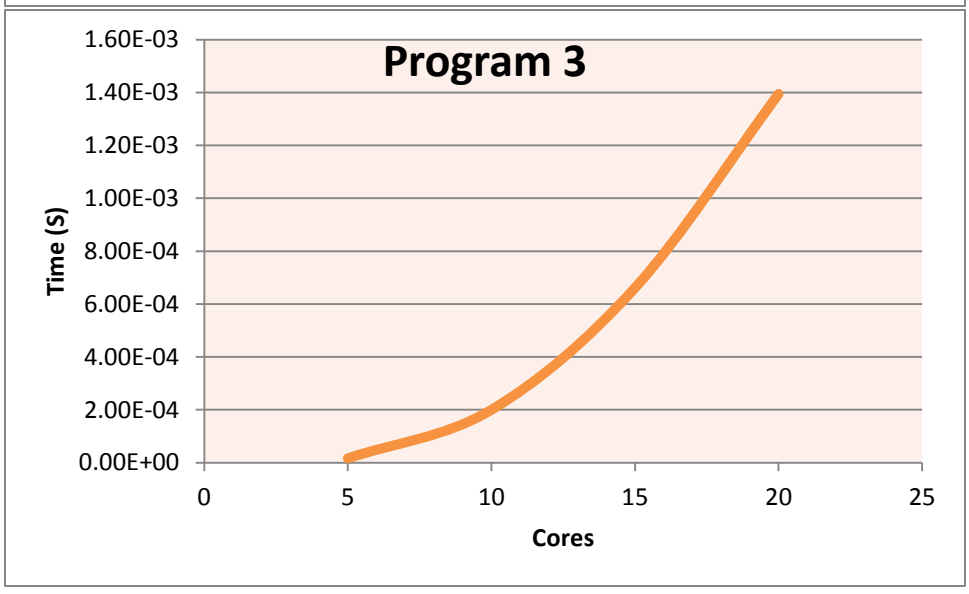
This table shows average time it took to find min and max at a given iteration.

I believe that the reason we don't see more of an increase in benefit in the third algorithm is due to the hardware running the program. Our machines in the lab don't have the hardware to actually run each node on a different core. I think that having to fake these extra cores on a few does not provide the right conditions to gain benefit from trying to leverage the cores to do more work.

Another interesting assumption I had challenged in my experiments was the behavior of the second algorithm. I thought since the number of messages it would have to send grows the fastest, it would be the most affected by an increase in cores. Instead we see the rate at which program two slows down actually decrease with more cores, while the others increased greatly.


Program 1
3.00E-04
2.50E-04
2.00E-04
1.50E-04
1.00E-04
5.00E-05
0.00E+00
Time(S)
0
5
10
15
20
25
Cores


Program 2
0.06
0.05
0.04
0.03
0.02
0.01
0
Time(S)
0
5
10
15
20
25
Cores


Program 3
1.60E-03
1.40E-03
1.20E-03
1.00E-03
8.00E-04
6.00E-04
4.00E-04
2.00E-04
0.00E+00
Time (S)
0
5
10
15
20
25
Cores

I cannot come up with a good explanation for this. It seems completely counterintuitive, and therefore I think it is likely due to an error in my calculations somewhere. The most likely culprit would be the placement of the timer in programs.

Concurrency is complex, and hard to get right. Measuring the performance of a concurrent program is just as complex. I've demonstrated here how what one thinks is going to happen isn't always what turns out. This time I think it's a bug in the program, but I should also question my assumptions, perhaps there is something going on at a much deeper level than programmer error.