PA2: Mandelbrot

Evan Grill

CS 415

March 14, 2017

Problem

This project asked us to calculate the following:

- How long it takes to calculate the Mandelbrot set sequentially.
- How long it takes to calculate the Mandelbrot set in parallel.
- Vary the parameters (Image size, number of parallel tasks) and see how that affects the performance.

Procedure

For calculating the Mandelbrot set, I used the "escape time" algorithm. This algorithm calculates the value for individual pixels up to a specified maximum iterations, which I set at 255. To divide the task up for parallel computation, I used the static approach and had the data divided up evenly between the slave tasks with the master task handling the return and copying of memory. This method has the added benefit that running the program with 2 parallel tasks is equivalent to running the program sequentially as all of the calculation is done on a single task.

Data

	Image Width (pixels)						
Tasks	**1000**	**2000**	**4000**	**8000**	**16000**	**24000**	**32000**
2	0.245	0.929	3.906	16.543	73.899	170.99	319.580
4	0.163	0.634	2.628	10.626	43.222	105.61	112.500
8	0.107	0.370	1.508	6.111	25.670	56.626	199.790
16	0.066	0.200	0.801	3.869	17.679	57.390	118.490
24	0.111	0.177	0.818	3.833	17.544	45.568	110.260

Table 1: Execution time (in seconds) of Mandelbrot set calculation based on variation of image size and level of parallelism.

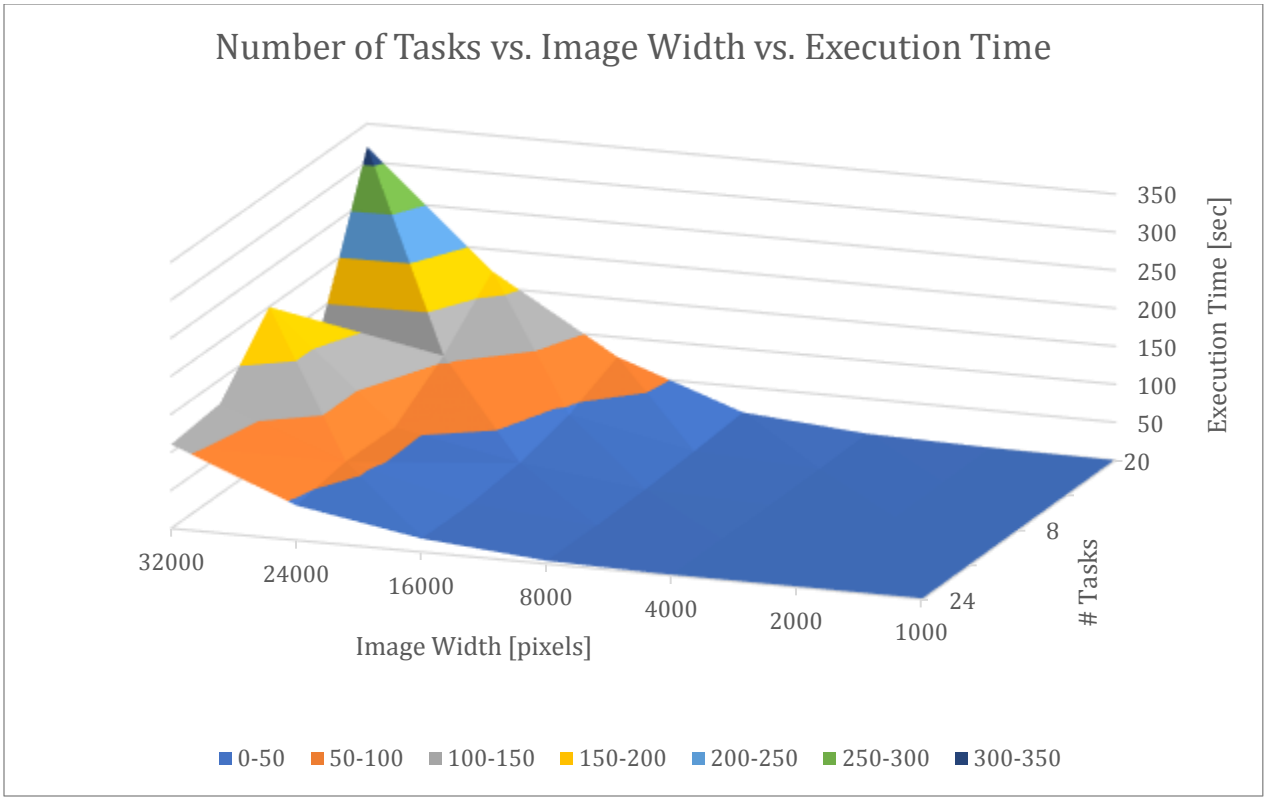

Figure 1: Chart representing the data from Table

	Image Width (pixels)						
Tasks	1000	2000	4000	8000	16000	24000	32000
2	1.000	1.000	1.000	1.000	1.000	1.000	1.000
4	1.503	1.465	1.486	1.557	1.710	1.619	2.841
8	2.290	2.511	2.590	2.707	2.879	3.020	1.600
16	3.712	4.645	4.876	4.276	4.180	2.979	2.697
24	2.207	5.249	4.775	4.316	4.212	3.752	2.898

Table 2: Speedup factor for parallel execution vs. sequential execution. ($S = {}^{t_s}/_{t_p}$) (2 tasks is equivalent to sequential execution.)

Results

For the sequential version of the algorithm (listed as 2 tasks), you see the results you'd expect with each increase in image width, you see a likewise increase in execution time. It's important to note that image size is listed by width, but you're increasing the size of the image by width * height pixels (I omitted the height because the image generated is square, but it need not be). Therefore, a doubling in image width leads to a quadrupling of both total pixels and execution time.

For the parallel version, the results show that increase in number of parallel tasks leads to a nearly linear speedup until you reach the larger size images or larger amounts of parallel tasks. For the image size issue, it can be speculated that increased message passing time would be the bottleneck.

For the increased parallel tasks, two issues come into play. The first is separation of the cores. The H1 cluster this data was collected from has eight cores per node. If the program is run with eight or less parallel tasks, the data all stays on a single node so communications time is very short. Once you break past the eight-task threshold, you are now splitting the computation across multiple nodes, and thus communication time increases dramatically. This happens again at each multiple of eight tasks. One particular value that is of note is the time speedup for a 1000x1000 pixel image on 24 cores. It's likely that message passing time is so much larger than the computation time that you reach an upper bound on the speedup factor.

One major issue I found while doing this project is that MPI is very specific about it's message passing. Every send and receive must be paired with a matching operation. Failure to do so will cause the program to go into deadlock when running MPI_Finalize(). The other issue I found was that while we were asked to collect data up to 32 cores, it appears that the H1 cluster is not configured, or we're not given the correct permissions, to allow a single program to request that many cores.