Scientific and Parallel Computing
Parallelisation of the Mandelbrot Set

Anis Jonischkeit

May 9, 2017

1 Introduction

This report will focus on the task of parallelising the Mandelbrot set. I will be discussing my implementation of two static task assignment approaches and one dynamic approach. This discussion will include relative implementation details, problems I encountered as well as observations and comparisons between the nonParallel, static parallel, and dynamic parallel implementations.

2 Implementation

For this assignment I have completed two versions of Task 1 and one version of Task 2.

2.1 Compiling and Running

2.1.1 Compiling

To compile the program, run: mpicc -lX11 PROGRAM_NAME.c -o PROGRAM_NAME -g -lm. This will allow for debugging and will link the math library properly

2.1.2 Running

To run the program, run: mpirun -hostfile YOUR_MACHINE_FILE -np NUMBER_OF_PROCESSORS mandelbrot BOUNDARY_LEVEL IMAGE_WIDTH SCALLING_FACTOR. Task2 has three additional parameters:

– STEP_SIZE: the number of rows that a single processor should process at a time

– INITIAL_STEP_SIZE: the number of rows that a single processor should process on a processors initial task

– DEBUG: should be the characters '-d'. Starts the program in debug mode. for this to work you must have a VSCode environment with the debug native extension and you must set your username and password for the server in the setupDebugging function.

NOTE: for all of the parallel programs, NUMBER_OF_PROCESSORS needs to be minimum 2

2.2 Task 1

2.2.1 Attempt 1

On my first attempt of Task 1 I decided to split the tasks evenly between the slaves so that each slave gets the $total\_amount\_of\_pixels/number\_of\_slaves$. So if there were 4 slaves then the first one would get the first quarter of image plane (split horizontaly not using recursive bisection)

I had the idea that the master shouldn't send out any data. Each processor could just work out what it needs to do based on its rank and an algorithm that they each follow. This would remove one receive from all of the slaves and number_of_slaves sends. The program that the master runs is just a for loop that has as many receives as there are slave processors. When master receives a buffer, it then draws the returned pixels to the screen translating the relative buffer positions to absolute image positions (the start of the buffer will always be position 0 but we don't want to draw to position i for each item of the returned buffer). There may be a few extra pixels that need to be drawn at then end. This happens if the number of pixels to be drawn does not divide evenly among the number of processors. If this is the case then the master will draw these at the end.

The slaves on the other hand will work out where the start and end of the chunk of pixels they need to process is. They will then work out whether each of their allocated pixels belong to the mandelbrot set. if the number of pixels to be drawn does not divide evenly among processors the first few processors will workout one extra pixel taken from the end of the image plane. After all pixels have been calculated, the pixel values are sent back to the master

2.2.2 Attempt 2

They biggest problem that I found with Attempt1 was that the top and bottom sections would get done really quickly if the Boundary Level increases. This happens since increasing the boundary level makes each processor do more calculations unless the specific pixel is clearly white. It is generally quite quick to work out if a pixel is white, however it is always slow to work out whether a pixel is black. The tasks in the middle would take a really long amount of time to finish (since there are more black pixels in the middle than on the outside). I for Task1Attempt2 decided to split the image plane so that every processor would have to process out every s^{th} row of pixels (where s is the number of slaves). This alone drastically increased the performance.

Another thing that is different is the fact that master now delegates the tasks that need to be done to the slaves (in preperation for dynamic task allocation) using MPI_Scatter. The master also holds on to which processors are doing what so that it can draw back to the right positions.

There is one major downside to the approach I chose for Attempt 2. Since master keeps track of what every processor does, if the imagewidth gets too big the size of the array which master uses just gets too big causing a crash. A more elegant solution is implemented in Task 2.

2.3 Task 2

I implemented Task to based on Task 1 Attempt 2 however I removed the big array that kept track of which row each processor was doing. Instead I replaced this with a very small state array which only keeps track of the first item of the last chunk that was allocated to a processor. I decided on making the program have two possible sizes for the chunks. There is an initial size which is only used as the size for the first chunk of each processor. All subsequent chunks are then of a different set size. Having there be an initial chunk size can help in speeding communication as we don't send as many tasks (the initial chunk size should quite big). The default size for the initial chunk is half of the total image size divided by the number of slaves. Sending off half of the processes straight away is a good idea because it is very unlikely that a processor will take twice as long as the average time taken for all other processors to complete the task using dynamic task allocation (especially since we are giving quite a spread of items in our chunks). So if rank 3 finishes late, if it still finishes its section before the rest of the task is processed, then it will only speed up the task. This cuts down communication a fair bit since we don't have to send as many different tasks (all of which would have the communication overhead).

After the first half of the data to be processed is sent of to the processors, dynamic task allocation occurs for the rest of the program. The dynamic task allocation goes as follows:

- The Master waits to receive a completed chunk from any slave
- A slave finishes processing tasks and sends back the tasks he just worked on
- The master takes those completed tasks and (if there are tasks left) allocates some new tasks to the slave. if no tasks are left a buffer starting with a value of -1 is returned to the slave
- The master works on drawing the completed task and waits to receive more completed tasks (if there are any tasks still sitting on a slaves)
- If there are no tasks left for the slaves to process, the master will send back a buffer with a value -1 as the first index (indicating to the slave that there is nothing left to do)

3 PCAM (Foster's Design Methodology)

While undertaking the task, we were instructed to use the PCAM methodology. It is quite difficult to design a parallel program if you don't have a logical methodology. In 1995, Ian Foster proposed the PCAM methodology, a four-stage design consisting of these processes:

1. Partitioning
2. Communication
3. Agglomeration
4. Mapping

3.1 Partitioning

The goal of the partitioning stage is to break down the problem to find as much parallelism as possible. This stage is the only stage in which we break down data and computations so we need to make sure to break down the problem as much as possible. There are two main ways in which we can break down our problem:

1. Domain Decomposition
2. Functional Decomposition

3.1.1 Domain Decomposition

Domain decomposition deals with decomposing the data into many small pieces to which parallel computations can be applied. In the case of the mandelbrot set, each point on the plane that needs to be calculated can be calculated independently of anything else. All we need to know about the particular point is the position of it on the plane. This makes the points on the plane perfect candidates for domain decomposition.

3.1.2 Functional Decomposition

Functional decomposition is the process of partitioning computations that need to be performed. I could only think of one example of functional parallelism for the task, that is the task of drawing the points to the XServer. Due to the limitation that only rank 0 can communicate to the XServer, this is an interesting observation but useless for our program implementation.

3.2 Communication

Once the program has been partitioned, we need to look at the communication that will occur between processors. In most parallel programs there needs to be some sort of communication since all processors are trying to achieve the same task. While communication is very useful, we must make sure to minimise communicate as much as is reasonable since it is generally far slower than computation.

3.2.1 Global Communication

A global communication operation is one in which many tasks must participate. My implementation of the parallel mandelbrot set uses global communication since all of the processors must talk to the master, they don't talk with one another. Although global communication can be slower since everyone has to wait for the master, a local communication approach would not have worked for the reason that all of the data has to come back to the master anyway to be drawn, so passing it along to a neighbour process would be completely unnecessary and a waste of communication.

3.2.2 Unstructured and Dynamic Communication

The dynamic version of the task Part2Attempt2 has both an unstructured and a dynamic part to it. The program works by first dividing half of the tasks between the slaves. The master slave will receive tasks from any process and send back tasks to processors that have finished their tasks.

3.2.3 Other Implementation Ideas

An idea I had started to play with was a divide and conquer strategy for dynamic task allocation. With this strategy, the master would assign tasks to two nodes, who would then assign further tasks to two more nodes each. This would eleviate the stress on rank0 to handle too many MPI_Sends at the same time. This approach however would still have the problem where only the master can draw to XServer so there would still be a delay of the master drawing and not receiving anything.

The solution for this problem would obviously be to make the processes asynchronous. I would have loved to have tried these options however with the time restrictions this sadly wasn't possible.

3.3 Agglomeration

Agglomeration deals with combining tasks in order to reduce communication. I was very mindful of this throughout the project. In Task1Attempt1 I decided to trade off replicated computation for reduced communication. I did this by not sending out any tasks to the processors and instead letting each processor deterimine for itself, what it needs to calculate (this is based on each processor's rank and the number of pixels to be processed). In the second attempt of task 1 I decided to get rid of this mechanism purely because it is only applicable to the first allocation (the non-dynamic). I decided that the extra time it would take to replicate this with the dynamic task allocation just wasn't worth it for the n-1 sends that would be saved (n is the number of processors). I also decided to not send tasks one by one (pixel by pixel or row by row). Instead I opted for a chunk of some size of rows to be sent.

3.4 Mapping

For Task1Attempt1, I used a bisection approach to split up the tasks. This approach split the image plane into as many sections as there are slave processors. Each section starts at:

$$(rank - 1) * \frac{pixels}{(size - 1)}$$

and ends at:

$$rank * \frac{pixels}{size - 1}$$

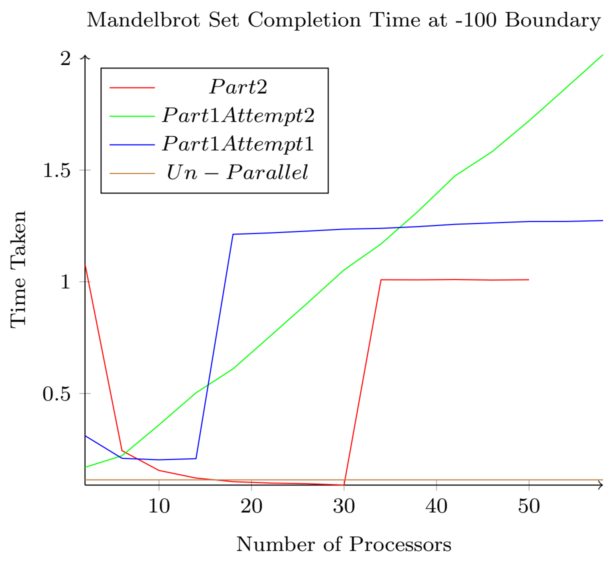
In Task1Attempt2 and Task2 the image plane was split by rows so that each processor would get allocated tasks following the pattern $[(row_i), (row_{i+1} + slaves), (row_{i+2} + 2 * slaves), (...), (row_{i+j} + j * slaves)]$ where i is the first row number to be processed and slaves is the number of slaves. In task1Attempt2 this was just done once with all of the rows (tasks), so all tasks were distributed in one go. In Task 2 however, this happens for half of the available rows. Then for the second half dynamic task allocation is used and each processor gets assigned a list of tasks following the same pattern (the amount of tasks that each processor gets allocated can be changed at execution time).

4 Analysis of Results

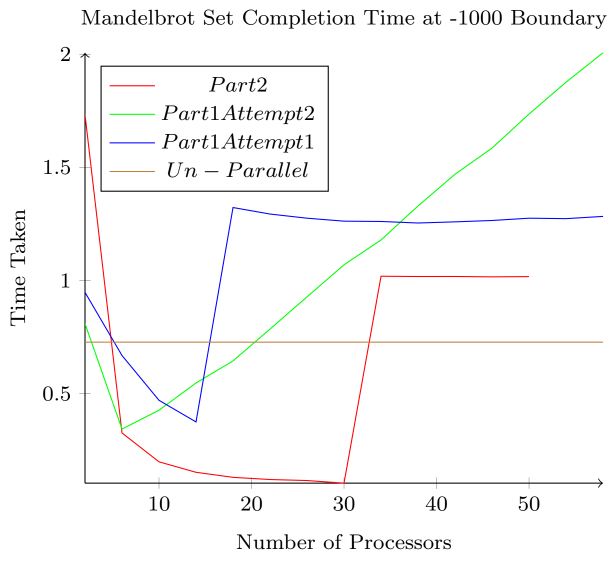
In order to get full accurate testing data, I wrote a python script that would run the Mandelbrot programs multiple times with different configurations set. To try to avoid outliers in our data, each configuration was run on a program three times and the median of those was taken (we take the median instead of the mean since we don't have a large enough sample size). The programs were only run three times per configuration, purely because of the fact that doing this already took roughly an hour and I didn't want to throttle the cluster for too long. The results of the runs can be seen below.

4.1 Runtimes for different numbers of processors and Boundary Levels

Boundary Level	Processors	Part 2	Part 1 Attempt 2	Part 1 Attempt 1	Un-Parallel
-100	2	1.079691	0.169912	0.311288	0.113702
-100	6	0.244176	0.221628	0.209951	0.113702
-100	10	0.155696	0.359769	0.20333	0.113702
-100	14	0.121732	0.503253	0.208478	0.113702
-100	18	0.105652	0.610299	1.212568	0.113702
-100	22	0.099352	0.756307	1.218644	0.113702
-100	26	0.096553	0.90322	1.226777	0.113702
-100	30	0.090209	1.052933	1.235544	0.113702
-100	34	1.0091	1.169359	1.239106	0.113702
-100	38	1.00861	1.315266	1.246816	0.113702
-100	42	1.010098	1.474082	1.257227	0.113702
-100	46	1.007772	1.582104	1.263341	0.113702
-100	50	1.009213	1.720295	1.269792	0.113702
-100	54	nan	1.868044	1.269989	0.113702
-100	58	nan	2.016207	1.273968	0.113702
-1000	2	1.732651	0.808844	0.947267	0.727143
-1000	6	0.325808	0.343067	0.668161	0.727143
-1000	10	0.198086	0.426369	0.470109	0.727143
-1000	14	0.151849	0.546753	0.374471	0.727143
-1000	18	0.129526	0.644327	1.322323	0.727143
-1000	22	0.119876	0.784904	1.293477	0.727143
-1000	26	0.11541	0.927412	1.27516	0.727143
-1000	30	0.103908	1.069196	1.262004	0.727143
-1000	34	1.0186	1.178855	1.26067	0.727143
-1000	38	1.017353	1.329361	1.253852	0.727143
-1000	42	1.017438	1.469555	1.258992	0.727143
-1000	46	1.015896	1.585052	1.265015	0.727143
-1000	50	1.016957	1.735604	1.275161	0.727143
-1000	54	nan	1.87646	1.27316	0.727143
-1000	58	nan	2.006554	1.283098	0.727143
-10000	2	8.091371	7.160035	7.366111	6.785325
-10000	6	1.574267	1.576296	5.248298	6.785325
-10000	10	0.874298	1.110733	3.312238	6.785325
-10000	14	0.611386	1.018151	2.397773	6.785325
-10000	18	0.485743	1.008288	2.885953	6.785325
-10000	22	0.404687	1.08191	2.586359	6.785325
-10000	26	0.344976	1.170039	2.35819	6.785325
-10000	30	0.295846	1.286601	2.170694	6.785325
-10000	34	0.468388	1.369677	2.113144	6.785325
-10000	38	0.651917	1.494727	2.001818	6.785325
-10000	42	0.441952	1.617285	1.911772	6.785325
-10000	46	0.621622	1.795757	1.853137	6.785325
-10000	50	0.613129	1.857626	1.803725	6.785325
-10000	54	nan	1.986203	1.763825	6.785325
-10000	58	nan	2.123963	1.746899	6.785325
-100000	2	71.615065	70.651134	71.418312	67.453196
-100000	6	14.068689	14.247951	50.9975	67.453196
-100000	10	7.774323	7.960181	31.728759	67.453196
-100000	14	5.366838	5.744237	22.610345	67.453196
-100000	18	4.173985	4.612883	18.48201	67.453196
-100000	22	3.523273	3.98738	15.482117	67.453196
-100000	26	2.949608	3.598065	13.192974	67.453196
-100000	30	2.571287	3.434617	11.269945	67.453196
-100000	34	2.235044	3.235731	10.612285	67.453196
-100000	38	2.040041	3.176472	9.474735	67.453196
-100000	42	1.800759	3.105086	8.565709	67.453196
-100000	46	1.765413	3.073391	7.964273	67.453196
-100000	50	1.604652	3.106167	7.385809	67.453196
-100000	54	nan	3.148719	6.973225	67.453196
-100000	58	nan	3.202683	6.742882	67.453196

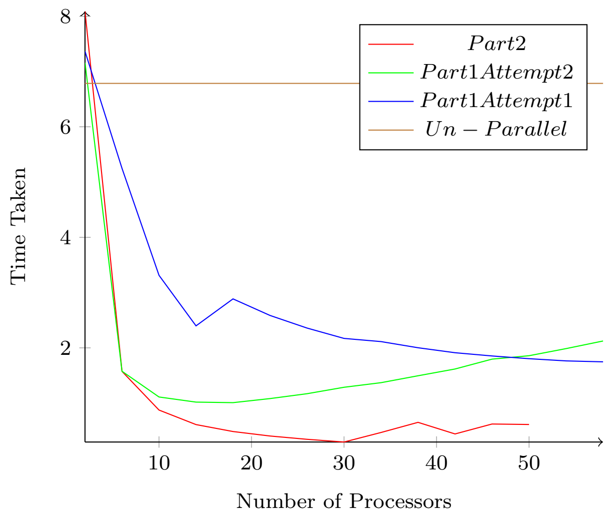


In the first line graph you can see that the Un-Parallel algorithm is actually the fastest for the majority of the time. This is Because there is so little information to process that communication slows the task down too much. When you have too small of a problem a parallel algorithm can just slow you down.



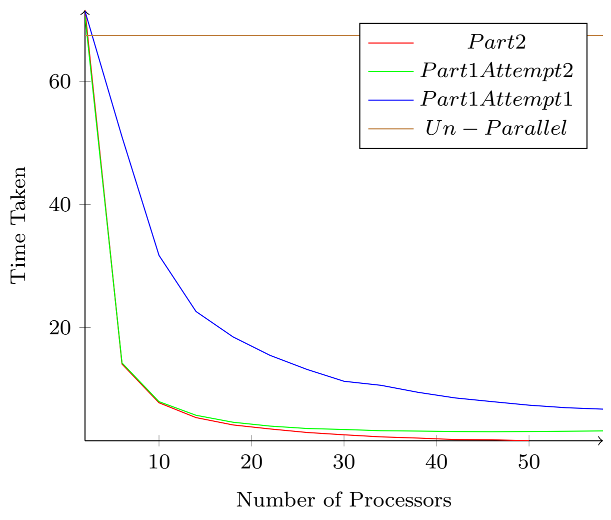
In this line graph you can start to see how the parallel algorithms are now overtaking the Un-Parallel algorithm initially. They do however become slower when too many processors are used (again due to there being too much communication for the size of the task).

Mandelbrot Set Completion Time at -10000 Boundary



Having now marginally increased the boundary level, you can clearly see that all of the Parallel programs outperform the Un-Parallel one. The fastest program quickly becomes the Dynamically allocated one. Interestingly though, Part1Attempt2 starts of being much quicker than Part1Attempt1 however slowly enough Part1Attempt1 overtakes Part1Attempt2. The reason why Part1Attempt2 is quicker initially is because the tasks are more evenly divided with the amount of work that needs to be done on them. When more processors are used however, the advantage that the evenly dividing does becomes a lot smaller. It in fact becomes small enough that the advantage of Part1Attempt1 not having to delegate tasks from the master to the slaves is enough to make it faster than Part1Attempt2.

Mandelbrot Set Completion Time at -100000 Boundary

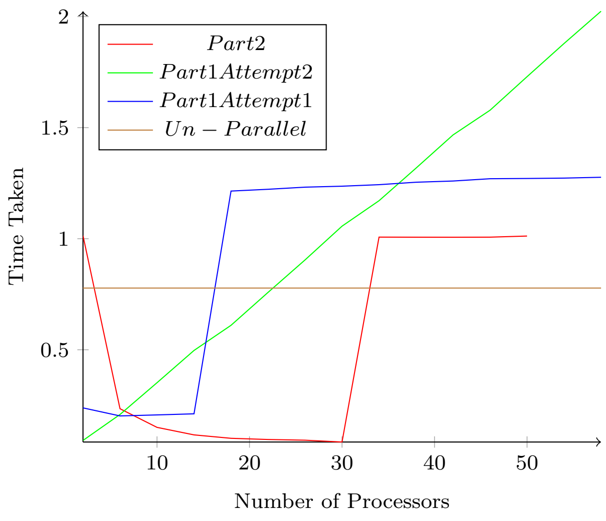


The pattern of the parallel programs leaving the Un-Parallel one behind continues on from the previous graphs.

4.2 Runtimes for different numbers of processors and Image Widths

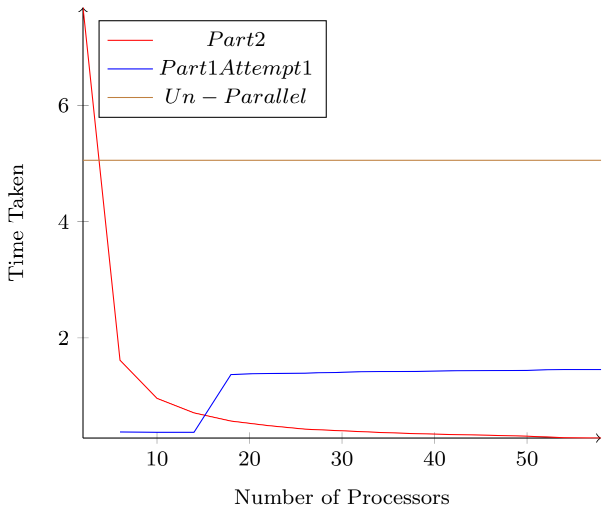
Image Width	Processors	Part 2	Part 1 Attempt 2	Part 1 Attempt 1	Un-Parallel
1000	2	1.013597	0.091846	0.238421	0.777827
1000	6	0.234642	0.20875	0.202216	0.777827
1000	10	0.150697	0.351689	0.206773	0.777827
1000	14	0.116866	0.496925	0.211699	0.777827
1000	18	0.101467	0.610154	1.214504	0.777827
1000	22	0.096099	0.758273	1.222531	0.777827
1000	26	0.092978	0.904456	1.232061	0.777827
1000	30	0.084732	1.056459	1.236403	0.777827
1000	34	1.007179	1.171057	1.243261	0.777827
1000	38	1.00682	1.317841	1.254286	0.777827
1000	42	1.006663	1.46721	1.259734	0.777827
1000	46	1.006987	1.578416	1.270033	0.777827
1000	50	1.012003	1.7294	1.271172	0.777827
1000	54	nan	1.878843	1.272844	0.777827
1000	58	nan	2.024149	1.276601	0.777827
2000	2	7.687204	nan	nan	5.057529
2000	6	1.616238	nan	0.381096	5.057529
2000	10	0.960089	nan	0.376527	5.057529
2000	14	0.710357	nan	0.3766	5.057529
2000	18	0.569654	nan	1.37125	5.057529
2000	22	0.492112	nan	1.389686	5.057529
2000	26	0.430539	nan	1.393154	5.057529
2000	30	0.402363	nan	1.409443	5.057529
2000	34	0.374518	nan	1.423018	5.057529
2000	38	0.353119	nan	1.424881	5.057529
2000	42	0.337642	nan	1.43293	5.057529
2000	46	0.325403	nan	1.439374	5.057529
2000	50	0.309601	nan	1.442381	5.057529
2000	54	0.284411	nan	1.457414	5.057529
2000	58	0.27688	nan	1.456773	5.057529
3000	2	nan	nan	nan	11.819257
3000	6	5.253235	nan	0.672533	11.819257
3000	10	3.024692	nan	0.642908	11.819257
3000	14	2.178846	nan	0.636423	11.819257
3000	18	1.714765	nan	1.624608	11.819257
3000	22	1.436928	nan	1.639275	11.819257
3000	26	1.261905	nan	1.646149	11.819257
3000	30	1.126943	nan	1.659537	11.819257
3000	34	1.031085	nan	1.684666	11.819257
3000	38	0.957293	nan	1.698809	11.819257
3000	42	0.878274	nan	1.71141	11.819257
3000	46	0.842676	nan	1.698746	11.819257
3000	50	0.796878	nan	1.731182	11.819257
3000	54	0.754636	nan	1.750715	11.819257
3000	58	0.723396	nan	1.757216	11.819257
4000	2	nan	nan	nan	33.797860
4000	6	12.254745	nan	nan	33.797860
4000	10	6.981008	nan	1.023242	33.797860
4000	14	4.972647	nan	1.004863	33.797860
4000	18	3.865553	nan	1.969293	33.797860
4000	22	3.21313	nan	1.9862	33.797860
4000	26	2.759769	nan	1.9936	33.797860
4000	30	2.44499	nan	2.004942	33.797860
4000	34	2.212623	nan	2.046816	33.797860
4000	38	2.035126	nan	2.061946	33.797860
4000	42	1.881858	nan	2.058307	33.797860
4000	46	1.753612	nan	2.105131	33.797860
4000	50	1.620229	nan	2.10329	33.797860
4000	54	1.538583	nan	2.126209	33.797860
4000	58	1.475184	nan	2.101269	33.797860

Mandelbrot Set Completion Time at 1000 Width

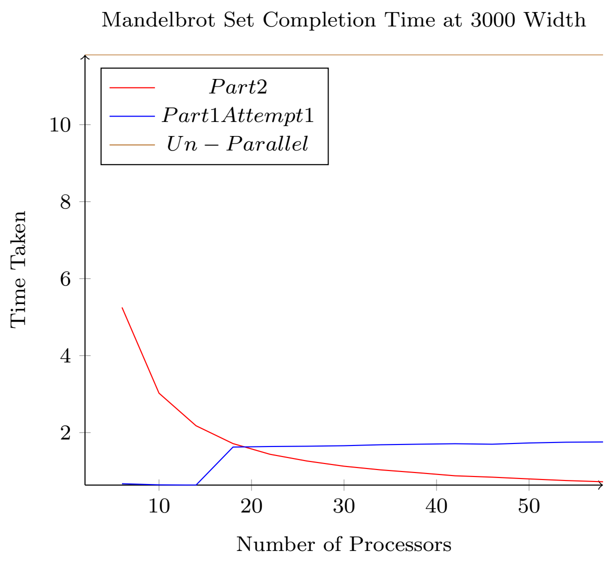


This time you can see straight away that the parallel algorithms are already predominantly ahead.

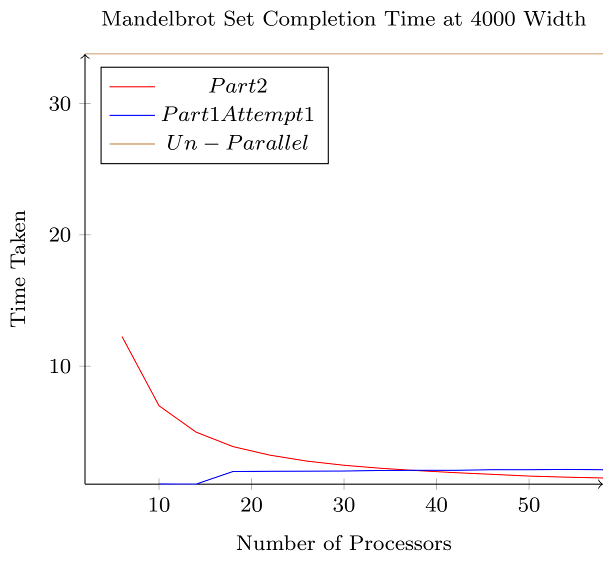
Mandelbrot Set Completion Time at 2000 Width



At this stage you can see that the Un-Parallel program is far behind both of the parallel programs. Notice also that Part1Attempt2 was not able to be drawn since too much memory was used by the program making it crash.



The Unparallel program gets even further behind both of the parallel programs.



4.3 Findings

One of the key things that I have learned from this is the fact that the communication can really be the difference between a faster and slower program. Notice how in the first few graphs the time taken goes down until it hits some point and then starts going back up. I believe that this is due to the larger and larger amounts of communication that is needed for handling more processors. You don't see the decrease of speed in the last two graphs, I do however believe that if we had more processors running the program, it would at some stage decreasing in speed.