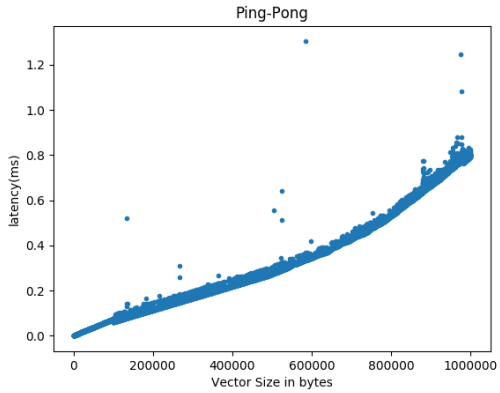
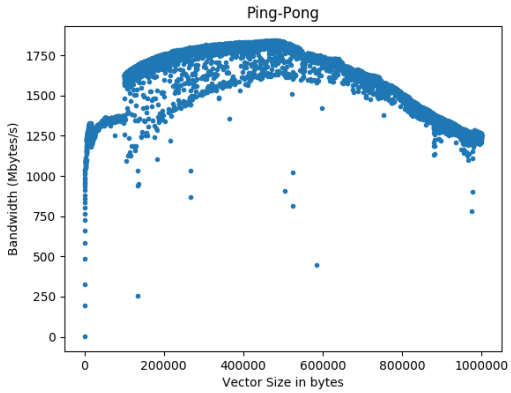
HPC I Homework 1

Morse, Michael October 5, 2017



(a) latency vs message size for ping pong



(b) Bandwidth vs message size for ping pong

Problem 1

My error check code is in the hw2 folder and is called by all codes in this assignment. For the error check code I had the error code looked up, printed to screen than I called MPI_ABORT. Therefore, the difference between my error handling the default is that mine will print look up the error code by then abort.

Problem 2

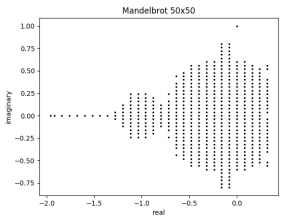
For the ping-pong problem I sent INTEGER(kind=8) 8 byte FORTRAN integers back and forth between two processors and had processor 0 time how long it took. I did this for array sizes 1 (8 byte) to one million (1 Mbyte). I did this in step size of 100 bytes. Unfortunately this resulted in a lot of noise in my plots. The latency is plotted vs message size where latency is half the time it took for the message to send and receive. I also plotted the bandwidth vs message size where bandwidth is calculated as message size over latency.

We can estimate a bandwidth of about 1500 Mbytes/second and a latency on the order of millisecond for a latency.

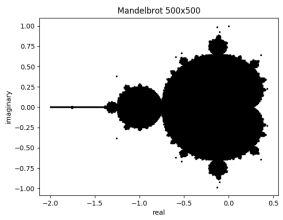
Problem 3

Serial

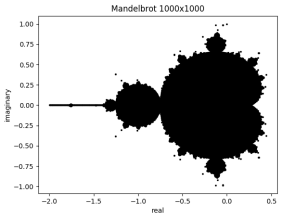
We see that as the grid size increases the area gets better until a grid of about 1000 by 1000 where is begins to level off.



(a) Course Mandelbrot set



(b) Medium Mandelbrot set



(c) Fine Mandelbrot set

Figure 2: Serial Mandelbrot sets for serial

MPI-row by row

For the first way of running the code in parallel I elected to send each processor a the a box with height 2 and width depending on the number of processors. As the load was not balanced in the case some processors were working harder then others. For example the ones at the end had very little work as the Mandelbrot set does not extend to $x = 2$.

As for area vs grid size we see the same trend as before, the area only gets marginally better after a grid size of about 1000.

As for time vs number of course, how the code was written each core was splitting their grid into nxn points so increasing the number of cores increased the accuracy of the area and point density but did not decrease the time. Instead it increased the time with the number of cores as the cores in heavy dense areas had much more work to do. A time saving approach would have been to take the grid and divide by number of cores squared so it scaled. I think this would have a decrease time with increased cores.

MPI-round robin

For the round robin approach I made rank 0 the master. Each slave would send a non-blocking request to let the master know it was ready for a new point. The master would distribute the work point wise instead of grid wise. This evenly distributed the work over all processors so the point density in any one sub grid did not matter.

Here we have the same observation for area but as the work is being evenly distributed across

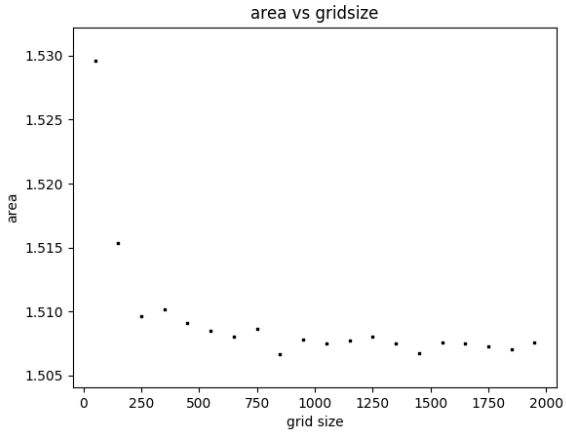
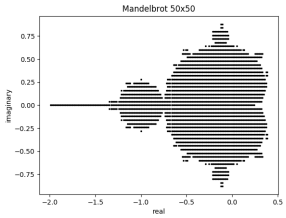
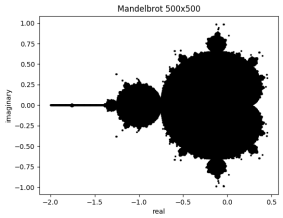


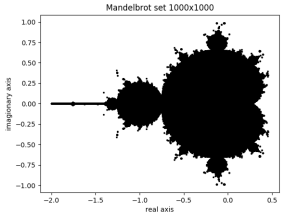
Figure 3: area vs grid size for serial



(a) Course Mandelbrot set



(b) Medium Mandelbrot set



(c) Fine Mandelbrot set

Figure 4: Row By Row Mandelbrot sets

all cores we also see that in increases the speed. From 4 cores to 8 cores we about half the time it takes, and by the time we are at 16 cores the time is about 30% of what it was at 4.

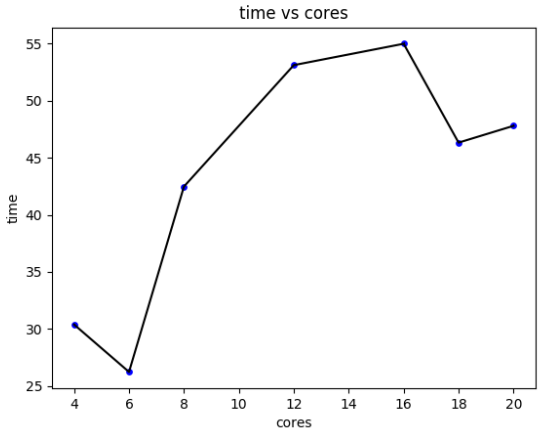


Figure 5: time vs number of cores for row by row

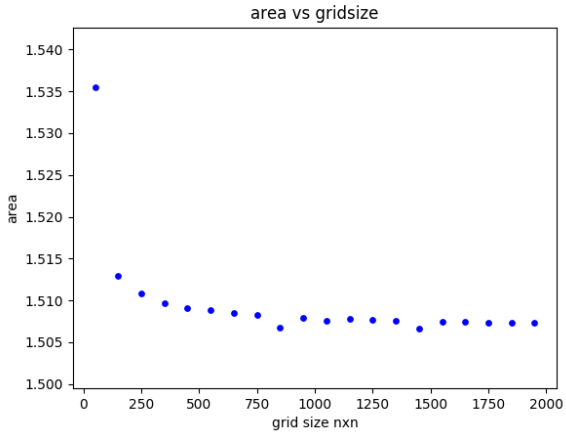
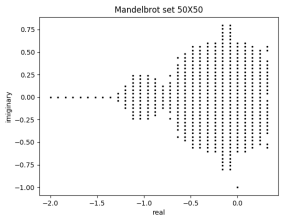
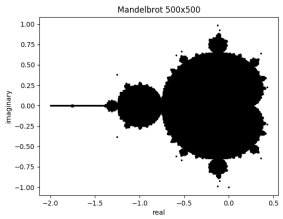


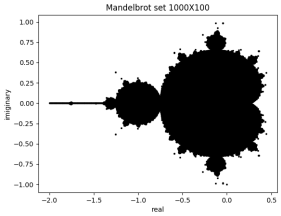
Figure 6: area vs grid size for row by row



(a) Course Mandelbrot set



(b) Medium Mandelbrot set



(c) Fine Mandelbrot set

Figure 7: Round Robin Mandelbrot sets

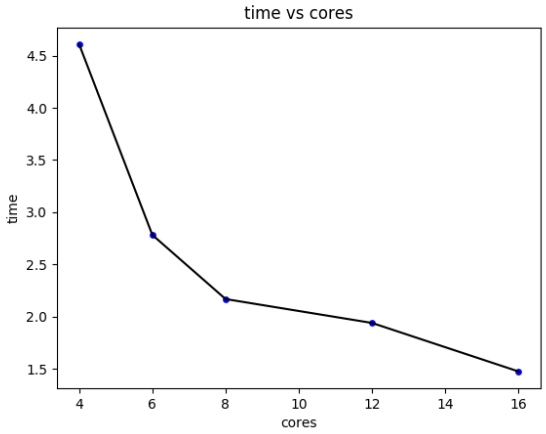


Figure 8: time vs number of cores for round robin

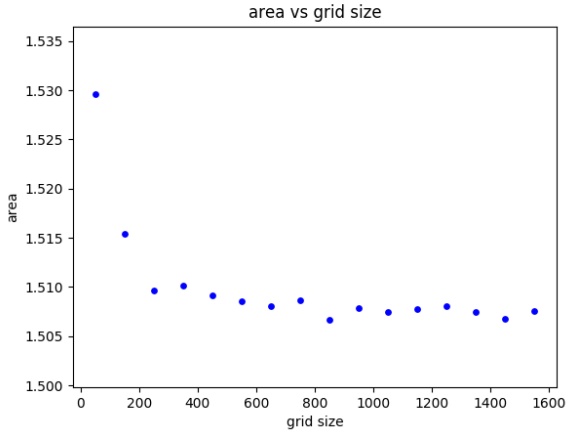


Figure 9: area vs grid size for round robin