**PA02: Mandelbrot Computations**

**Complex, Independent Computations Sequentially**

**vs. Static and Dynamic Parallel Modalities**

**Terence Henriod**

**Dr. Fred Harris**

**CS651: Parallel Computing**

**Monday March 10, 2014**

**Introduction**

Some tasks require many complex computations. Often such computations are independent of one another, so performing the computations one at a time, one after another, is less than ideal. If the computations can be performed alongside one another, i.e. in parallel, we can perform the computations in amounts of time that are more reasonable to humans. It should also be noted that different parallel computation modalities exist, two of which are explored here: static and dynamic job allocation.

**Theory**

*Mandelbrot Set*

In this report, we use the example of the Mandelbrot set. The Mandelbrot set is a set of points that comprise a boundary that creates a fractal image. This image is seen in pop-math, and requires (relatively) extensive computation to produce each point (or pixel) in the set (image). In short, each Mandelbrot point/pixel color/shade is found by computing the following formula iteratively, using the result in the previous iteration to prime the next:

|  |  |
| --- | --- |
|  | [1] |

where *zk* is the *kth* iteration of the formula and *c* is the complex value representing the location of the pixel in the complex plane.The computation stops after either a predetermined number of iterations or when it can be seen that the formula will diverge, that is, if the formula will produce a sequence of zeroes or progress to . The computation and theory of the Mandelbrot set is not the focus of this report, so it will not be discussed further.

*Speedup*

When computing things in parallel, it is advantageous to know how much faster the parallel algorithm/program runs compared to a sequential one. This is one metric used to evaluate the quality of a parallel algorithm, the *speedup factor*. The speedup factor represents how many times faster the parallel algorithm using *p* processors is compared to the sequential one. The speedup factor, *S(p)*, is computed as follows:

|  |  |
| --- | --- |
|  | [2] |

where *p* is the number of processors used in the parallel algorithm, *ts* and *tp* are the respective running times for sequential and parallel runs of the algorithms, and *f* is the fraction of the work in a program that must be run sequentially. The second and third expressions are known as Amdahl’s law. Ideally, the speedup factor will represent a number of factors , but this is often not the case due to various factors including non-parallelizable segments of a job or inter-process communication overhead. If *Sp* is small relative to *p*  or even approaches 1, then the parallel algorithm should either be improved or not used. Should *Sp* ever exceed *p*, this is known as *super-linear speedup*. A super-linear speedup factor is often the product of the use of a runtime that resulted from a sub-optimal sequential algorithm.

*Efficiency*

Parallel algorithms can also be measured in terms of how well the time to run a program is used by all of the processors. Efficiency measures how well the processors are used (what percentage of the runtime the processors spend working). Efficiency is found by:

|  |  |
| --- | --- |
|  | [3] |

where *ts*, *tp*, and *S(p)* are defined as in [2].

**Pseudo-Code**

It should be noted that the following pseudo-code should not be considered functional code. Functional code will be attached in the report.

*Sequential Pseudo-Code*

In a sequential program, it can be assumed that the work of a program is run as though it were only being run on a single processor. Thus operations are performed one after another, with no apparent re-ordering or concurrency.

double createMandelBrotImage( size )

{

run\_time

Image\_Matrix

startTimer()

// compute the pixels/values of each row

for( i = 0; i < size; i = i + 1 )

{

for( j = 0; j < size; j = j + 1 )

{

Image\_Matrix[i][j] = calcMandelPixel( i, j )

}

}

run\_time = stopTimer()

createMandelbrotImage( Image\_Matrix, size )

return run\_time

}

*Parallel Psuedo-Code*

In general, parallel programs have a *master* processor to direct the work and multiple *slave* processors to do most of the actual work. The master primarily performs communication and status checking operations, while the slaves primarily perform computations and only minimal sending of the results to the master.

*Static Job Allocation*

In a static job allocation algorithm each slave is allocated a set amount of work. In this algorithm, the slaves know which rows to compute based on their own number and the number of slaves. The slaves send the computation results to the master compiles the data.

Master:

double staticMaster( size )

{

run\_time

Image\_Matrix

slave\_id

rows\_left\_to\_compute = size

row\_to\_receive

startTimer()

while( rows\_left\_to\_compute > 0 )

{

receive( row\_to\_receive, slave\_id )

receive( Image\_Matrix[row\_to\_receive], slave\_id )

rows\_left\_to\_compute = rows\_left\_to\_compute - 1

}

run\_time = stopTimer()

}

Slave:

void staticSlave( size, slave\_id, num\_slaves )

{

row\_buffer

current\_row

current\_row = slave\_id - 1

while( current\_row < size )

{

for( j = 0; j < size; j = j + 1 )

{

row\_buffer[j] = calcMandelPixel( current\_row, j )

}

send( row\_buffer, master, slave\_id )

current\_row = current\_row + num\_slaves

}

}

*Dynamic Job Allocation*

With dynamic job allocation, slaves are given a segment of work, and are to report for more work once their segment is complete. In my scheme, work is apportioned to slaves in blocks of 5 rows. Again, the master does administrative work, compiling data and delegating new work.

Master:

double dynamicMaster( size, num\_slaves )

{

run\_time

Image\_Matrix

slave\_id

next\_row\_to\_compute

num\_received = 0

row\_recieved

next\_row\_to\_compute = ( num\_slaves – 1 ) \* size

startTimer()

while( num\_received < size )

{

receive( row\_received, slave\_id )

for( k = 0; k < size; k = k + 1 )

{

receive( Image\_Matrix[row\_received + k], slave\_id )

}

send( next\_row\_to\_compute, slave\_id )

next\_row\_to\_compute = next\_row\_to\_compute + 10

}

run\_time = stopTimer()

}

Slave:

void staticSlave( size, slave\_id )

{

row\_buffer

current\_row

current\_row = ( slave\_id – 1 ) \* 10

while( current\_row < size )

{

for( i = 0; i < 5; i = i + 1 )

{

for( j = 0; j < size; j = j + 1 )

{

row\_buffers[i][j] = calcMandelPixel(current\_row + i, j )

}

}

send( current\_row, master, slave\_id )

for( i = 0; i < 5; i = i + 1 )

{

send( row\_buffers[i], master, slave\_id )

}

receive( current\_row, master )

}

}

**Output Data Summaries**

The following tables and figures are representative of the output of the trial runs of the Mandelbrot set computation runs, listed by appropriate category. This is pretty boring and monotonous, but accurate, so if Devyani wanted to skip it, she could and still be confident that all the necessary items were here.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Computing Nodes Used in Experiment | | | | | | | |
| 0: | compute-2-14.local | 5: | compute-3-0.local | 10: | compute-1-21.local | 15: | compute-3-11.local |
| 1: | compute-2-14.local | 6: | compute-2-22.local | 11: | compute-41-10.local | 16: | compute-3-11.local |
| 2: | compute-2-14.local | 7: | compute-2-22.local | 12: | compute-3-5.local | 17: | compute-3-11.local |
| 3: | compute-3-0.local | 8: | compute-2-22.local | 13: | compute-3-11.local | 18: | compute-2-21.local |
| 4: | compute-3-0.local | 9: | compute-41-7.local | 14: | compute-3-11.local | 19: | compute-2-21.local |

Table 1: The processor nodes used for the Mandelbrot computation runs. If a run used *n* processors, then processors *n* through *n – 1* were used.

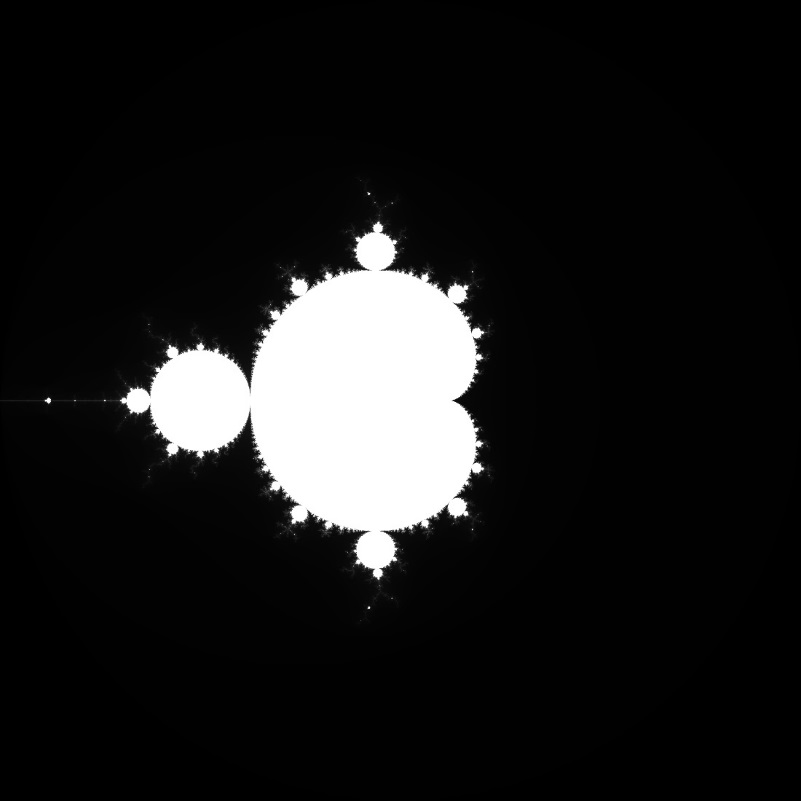


Figure 1: a 10,000 x 10,000 pixel Mandelbrot image in grayscale produced by a sequential algorithm. Note: this image was converted from .pgm format to .jpeg format for Windows compatibility.

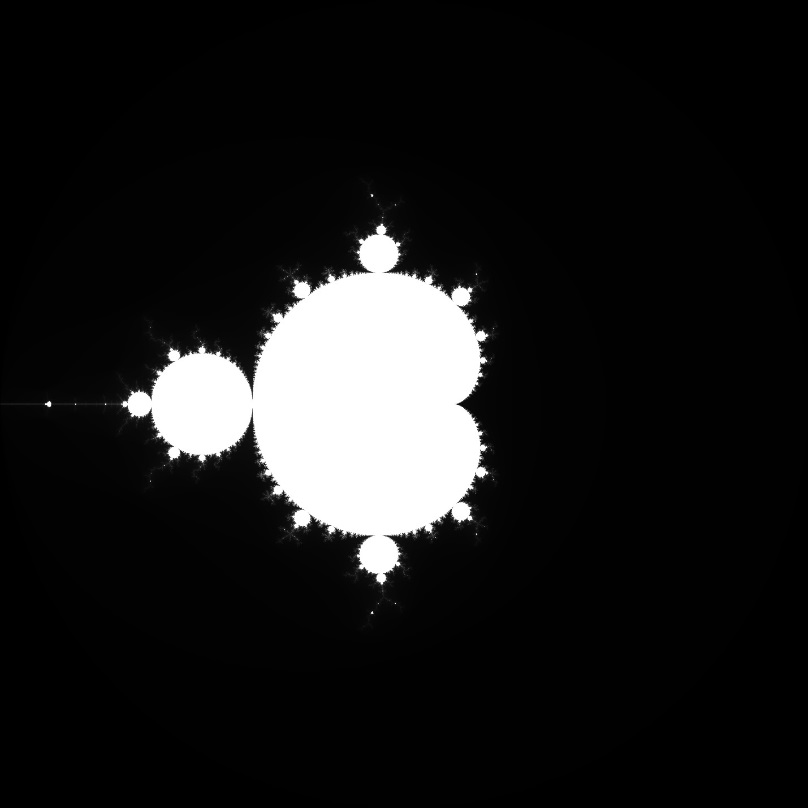


Figure 2: a 10,000 x 10,000 pixel Mandelbrot image in grayscale produced by a static job allocation parallel algorithm. Note: this image was converted from .pgm format to .jpeg format for Windows compatibility.

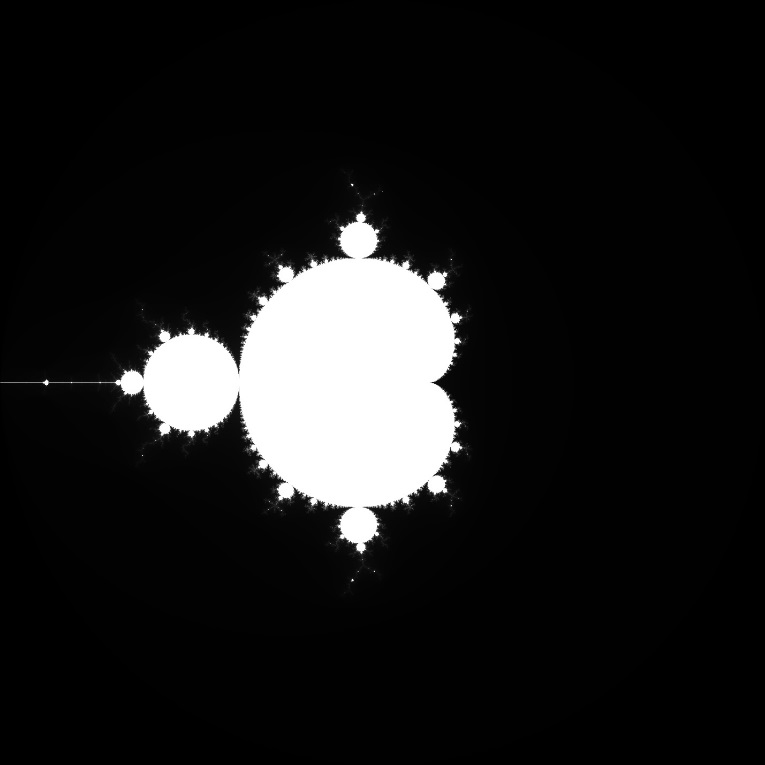


Figure 3: a 10,000 x 10,000 pixel Mandelbrot image in grayscale produced by a dynamic job allocation parallel algorithm. Note: this image was converted from .pgm format to .jpeg format for Windows compatibility.

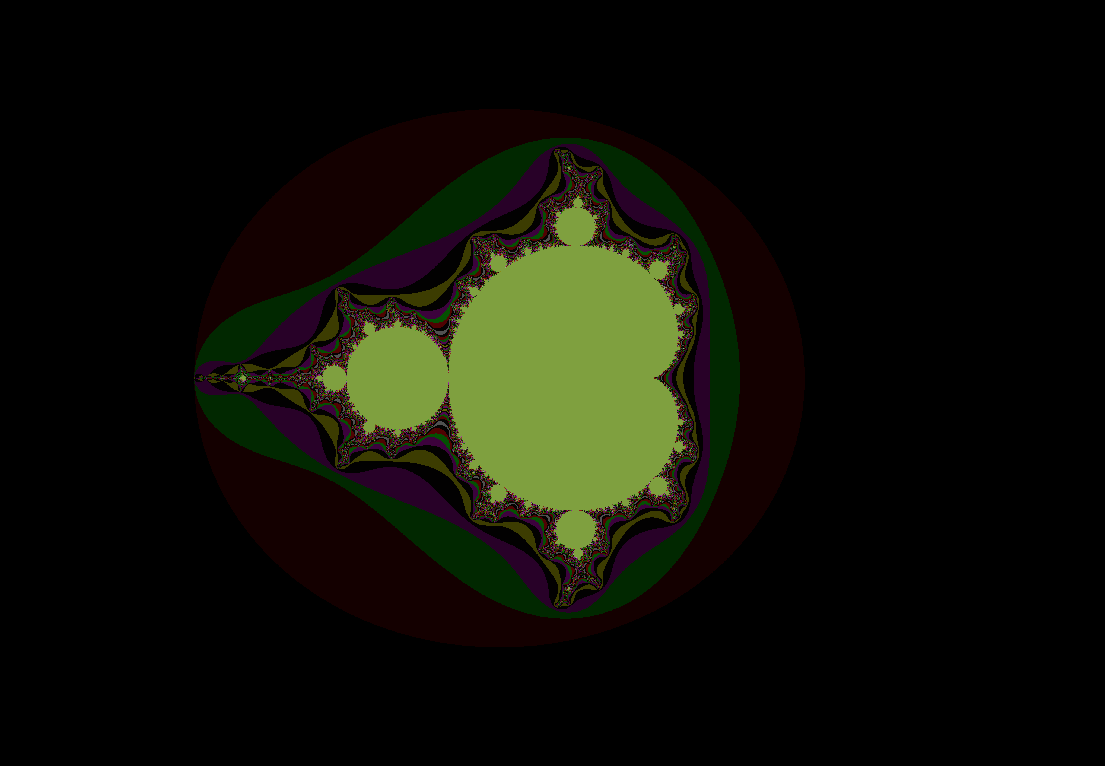


Figure 4: a 3,000 x 3,000 pixel Mandelbrot image in color produced by a sequential computation run. A smaller image was used and a screen shot was taken in order to get this due to the limitations of my personal laptop and Windows. A real, 10,000 x 10,000 .ppm color image will be included with the other submitted files.

Figure 5: The trend of the time required to compute images of various sizes sequentially.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Image Size (pixels) | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| Running Time (s) | 0.147 | 0.562 | 1.100 | 1.909 | 2.975 | 4.274 | 5.755 | 7.484 | 9.562 | 11.791 |
| Image Size (pixels) | 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 | 10000 |
| Running Time (s) | 14.132 | 16.869 | 19.675 | 22.782 | 26.287 | 29.815 | 33.754 | 37.812 | 42.082 | 46.507 |

Table 2: The benchmark average job computation times for computing Mandelbrot images of various sizes sequentially. These will act as the baselines for the later computations of speedup and efficiency.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Static Work Allocation Running Times | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 0.159 | 0.108 | 0.063 | 0.048 | 0.037 | 0.032 | 0.029 | 0.026 | 0.145 | 0.023 |
| 1000 | 0.501 | 0.244 | 0.176 | 0.194 | 0.126 | 0.257 | 0.248 | 0.222 | 0.239 | 0.338 |
| 1500 | 1.128 | 0.501 | 0.284 | 0.320 | 0.237 | 0.269 | 0.249 | 0.288 | 0.241 | 0.465 |
| 2000 | 1.907 | 0.795 | 0.479 | 0.441 | 0.327 | 0.344 | 0.322 | 0.412 | 0.302 | 0.288 |
| 2500 | 2.992 | 1.102 | 0.691 | 0.613 | 0.565 | 0.583 | 0.595 | 0.376 | 0.310 | 0.492 |
| 3000 | 4.283 | 1.503 | 0.932 | 0.727 | 1.025 | 0.737 | 0.523 | 0.489 | 0.423 | 0.553 |
| 3500 | 5.788 | 2.009 | 1.246 | 0.974 | 0.874 | 0.727 | 0.623 | 0.624 | 0.803 | 0.674 |
| 4000 | 7.523 | 2.580 | 1.592 | 1.397 | 1.006 | 0.973 | 0.966 | 0.752 | 0.632 | 0.638 |
| 4500 | 9.520 | 3.282 | 2.035 | 1.543 | 1.405 | 1.110 | 1.176 | 0.870 | 0.884 | 0.689 |
| 5000 | 11.693 | 3.999 | 2.416 | 2.025 | 1.584 | 1.347 | 1.365 | 1.104 | 1.066 | 0.841 |
| 5500 | 14.140 | 4.808 | 2.964 | 2.180 | 2.182 | 1.616 | 1.428 | 1.230 | 1.119 | 0.996 |
| 6000 | 16.800 | 5.762 | 3.474 | 2.802 | 2.276 | 1.922 | 1.613 | 1.409 | 1.316 | 1.117 |
| 6500 | 19.650 | 6.666 | 4.026 | 3.076 | 2.687 | 2.132 | 1.916 | 1.614 | 1.591 | 1.397 |
| 7000 | 22.791 | 7.704 | 4.708 | 3.627 | 3.058 | 2.525 | 2.263 | 2.098 | 1.681 | 1.547 |
| 7500 | 26.299 | 8.876 | 5.334 | 4.063 | 3.597 | 2.891 | 2.542 | 2.238 | 1.974 | 1.697 |
| 8000 | 29.755 | 10.068 | 6.045 | 4.811 | 4.094 | 3.356 | 2.820 | 2.479 | 2.281 | 1.828 |
| 8500 | 33.740 | 11.340 | 6.821 | 5.288 | 4.619 | 3.680 | 3.169 | 2.747 | 2.565 | 2.118 |
| 9000 | 37.541 | 12.704 | 7.785 | 5.848 | 4.996 | 4.223 | 3.502 | 3.564 | 2.802 | 2.644 |
| 9500 | 41.899 | 14.146 | 8.523 | 6.399 | 5.735 | 4.782 | 4.177 | 3.453 | 3.076 | 2.875 |
| 10000 | 46.626 | 15.725 | 9.478 | 7.211 | 6.107 | 5.068 | 4.214 | 3.915 | 3.418 | 3.126 |

Table 3: The average run times for static work allocation computation times by number of slaves used and image size.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dynamic Work Allocation Running Times | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 0.147 | 0.074 | 0.101 | 0.227 | 0.049 | 0.085 | 0.046 | 0.027 | 0.079 | 0.063 |
| 1000 | 0.513 | 0.213 | 0.307 | 0.259 | 0.103 | 0.153 | 0.121 | 0.136 | 0.295 | 0.309 |
| 1500 | 1.093 | 0.417 | 0.450 | 0.298 | 0.238 | 0.193 | 0.190 | 0.215 | 0.329 | 0.401 |
| 2000 | 1.912 | 0.680 | 0.606 | 0.380 | 0.377 | 0.282 | 0.265 | 0.254 | 0.410 | 0.621 |
| 2500 | 2.969 | 1.034 | 1.287 | 0.586 | 0.430 | 0.375 | 0.378 | 0.344 | 0.340 | 1.078 |
| 3000 | 4.220 | 1.475 | 1.306 | 0.802 | 0.713 | 0.641 | 0.705 | 0.531 | 0.609 | 1.934 |
| 3500 | 5.753 | 1.944 | 1.680 | 1.159 | 1.466 | 1.118 | 0.905 | 1.482 | 0.719 | 2.420 |
| 4000 | 7.553 | 2.511 | 2.019 | 1.531 | 2.306 | 1.558 | 1.664 | 1.416 | 1.417 | 3.328 |
| 4500 | 9.482 | 3.146 | 2.239 | 1.909 | 3.238 | 2.568 | 2.261 | 1.695 | 1.959 | 3.982 |
| 5000 | 11.745 | 3.918 | 2.639 | 2.501 | 3.732 | 2.821 | 3.197 | 2.708 | 2.927 | 4.122 |
| 5500 | 14.224 | 4.684 | 3.393 | 4.065 | 4.645 | 4.093 | 3.744 | 2.911 | 2.504 | 5.528 |
| 6000 | 16.777 | 5.517 | 4.541 | 4.542 | 5.423 | 4.748 | 4.656 | 4.109 | 3.706 | 6.328 |
| 6500 | 19.752 | 6.500 | 4.263 | 5.717 | 5.800 | 4.950 | 5.053 | 4.466 | 5.184 | 8.222 |
| 7000 | 22.683 | 7.539 | 4.973 | 6.144 | 7.016 | 5.743 | 6.049 | 5.446 | 5.070 | 9.867 |
| 7500 | 26.299 | 10.590 | 5.860 | 7.087 | 6.815 | 7.720 | 6.457 | 5.884 | 6.262 | 9.255 |
| 8000 | 29.861 | 10.590 | 6.378 | 9.955 | 8.018 | 7.512 | 6.599 | 6.626 | 6.135 | 10.773 |
| 8500 | 33.679 | 12.442 | 7.406 | 11.580 | 8.574 | 8.144 | 7.566 | 6.809 | 7.394 | 11.023 |
| 9000 | 37.651 | 13.166 | 8.121 | 11.613 | 9.900 | 9.404 | 8.525 | 8.073 | 8.015 | 12.770 |
| 9500 | 41.916 | 14.293 | 9.835 | 10.817 | 10.917 | 13.340 | 10.729 | 8.980 | 9.795 | 15.218 |
| 10000 | 46.735 | 15.855 | 9.925 | 11.457 | 12.448 | 11.035 | 12.430 | 9.845 | 10.568 | 15.861 |

Table 4: The average run times for dynamic work allocation computation times by number of slaves used and image size.

**Speedup and Efficiency**

Figures and tables representative of the Speedup and Efficiency results are displayed in table and graphical format in this section. It should be noted that due to ineffective message passing on the grid, some results are not as expected, the dynamic work allocation results in particular. However, speedup of 8 was achieved for each job type after reaching an appropriate number of processing nodes/slaves.

Figure 6: A graph representing the speedup of the static Mandelbrot computation run averages.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Static Job Allocation Speedups | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 0.925 | 1.362 | 2.327 | 3.041 | 3.972 | 4.529 | 5.038 | 5.560 | 1.010 | 6.471 |
| 1000 | 1.123 | 2.306 | 3.194 | 2.897 | 4.476 | 2.191 | 2.265 | 2.528 | 2.349 | 1.662 |
| 1500 | 0.975 | 2.195 | 3.877 | 3.438 | 4.648 | 4.082 | 4.411 | 3.818 | 4.572 | 2.364 |
| 2000 | 1.001 | 2.401 | 3.984 | 4.333 | 5.846 | 5.542 | 5.932 | 4.630 | 6.323 | 6.634 |
| 2500 | 0.994 | 2.701 | 4.309 | 4.855 | 5.267 | 5.102 | 4.997 | 7.907 | 9.588 | 6.052 |
| 3000 | 0.998 | 2.844 | 4.584 | 5.879 | 4.171 | 5.803 | 8.176 | 8.736 | 10.095 | 7.725 |
| 3500 | 0.994 | 2.865 | 4.620 | 5.910 | 6.585 | 7.914 | 9.233 | 9.228 | 7.171 | 8.536 |
| 4000 | 0.995 | 2.901 | 4.702 | 5.355 | 7.436 | 7.690 | 7.747 | 9.946 | 11.839 | 11.725 |
| 4500 | 1.004 | 2.914 | 4.698 | 6.197 | 6.805 | 8.618 | 8.134 | 10.988 | 10.821 | 13.881 |
| 5000 | 1.008 | 2.949 | 4.880 | 5.822 | 7.446 | 8.754 | 8.640 | 10.678 | 11.058 | 14.021 |
| 5500 | 0.999 | 2.939 | 4.768 | 6.483 | 6.477 | 8.747 | 9.898 | 11.490 | 12.632 | 14.185 |
| 6000 | 1.004 | 2.928 | 4.855 | 6.021 | 7.413 | 8.776 | 10.459 | 11.971 | 12.818 | 15.105 |
| 6500 | 1.001 | 2.951 | 4.887 | 6.396 | 7.323 | 9.228 | 10.267 | 12.187 | 12.365 | 14.086 |
| 7000 | 1.000 | 2.957 | 4.839 | 6.282 | 7.451 | 9.021 | 10.065 | 10.857 | 13.551 | 14.729 |
| 7500 | 1.000 | 2.962 | 4.928 | 6.469 | 7.308 | 9.093 | 10.342 | 11.745 | 13.316 | 15.491 |
| 8000 | 1.002 | 2.961 | 4.932 | 6.197 | 7.282 | 8.884 | 10.573 | 12.027 | 13.068 | 16.315 |
| 8500 | 1.000 | 2.976 | 4.948 | 6.383 | 7.308 | 9.171 | 10.652 | 12.287 | 13.158 | 15.939 |
| 9000 | 1.007 | 2.976 | 4.857 | 6.465 | 7.569 | 8.953 | 10.798 | 10.608 | 13.494 | 14.301 |
| 9500 | 1.004 | 2.975 | 4.937 | 6.577 | 7.338 | 8.800 | 10.075 | 12.189 | 13.679 | 14.639 |
| 10000 | 0.997 | 2.958 | 4.907 | 6.449 | 7.615 | 9.176 | 11.036 | 11.880 | 13.608 | 14.880 |

Table 5: A table representing the speedup of the static Mandelbrot computation run averages.

Figure 7: A graph representing the speedup of the dynamic Mandelbrot computation run averages. Do note that the required speedup of 8 was achieved, if not by the majority of runs. I suspect this lack of speedup is due to the volatility of the grid communications.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dynamic Job Allocation Speedups | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 1.002 | 1.979 | 1.449 | 0.647 | 3.023 | 1.736 | 3.208 | 5.508 | 1.848 | 2.320 |
| 1000 | 1.096 | 2.644 | 1.830 | 2.172 | 5.460 | 3.669 | 4.629 | 4.125 | 1.903 | 1.818 |
| 1500 | 1.006 | 2.635 | 2.442 | 3.693 | 4.621 | 5.688 | 5.798 | 5.125 | 3.348 | 2.746 |
| 2000 | 0.998 | 2.808 | 3.149 | 5.023 | 5.068 | 6.773 | 7.200 | 7.509 | 4.659 | 3.072 |
| 2500 | 1.002 | 2.878 | 2.313 | 5.073 | 6.917 | 7.933 | 7.867 | 8.637 | 8.751 | 2.761 |
| 3000 | 1.013 | 2.898 | 3.273 | 5.330 | 5.995 | 6.669 | 6.063 | 8.047 | 7.021 | 2.210 |
| 3500 | 1.000 | 2.961 | 3.427 | 4.967 | 3.926 | 5.149 | 6.362 | 3.884 | 8.004 | 2.378 |
| 4000 | 0.991 | 2.981 | 3.706 | 4.887 | 3.246 | 4.802 | 4.499 | 5.286 | 5.282 | 2.249 |
| 4500 | 1.008 | 3.039 | 4.271 | 5.009 | 2.953 | 3.723 | 4.229 | 5.640 | 4.882 | 2.401 |
| 5000 | 1.004 | 3.010 | 4.469 | 4.715 | 3.159 | 4.179 | 3.688 | 4.354 | 4.029 | 2.861 |
| 5500 | 0.994 | 3.017 | 4.165 | 3.477 | 3.042 | 3.453 | 3.775 | 4.854 | 5.644 | 2.557 |
| 6000 | 1.005 | 3.058 | 3.714 | 3.714 | 3.110 | 3.552 | 3.623 | 4.105 | 4.551 | 2.666 |
| 6500 | 0.996 | 3.027 | 4.615 | 3.442 | 3.392 | 3.975 | 3.894 | 4.406 | 3.795 | 2.393 |
| 7000 | 1.004 | 3.022 | 4.581 | 3.708 | 3.247 | 3.967 | 3.766 | 4.183 | 4.493 | 2.309 |
| 7500 | 1.000 | 2.482 | 4.486 | 3.709 | 3.858 | 3.405 | 4.071 | 4.467 | 4.198 | 2.840 |
| 8000 | 0.998 | 2.815 | 4.675 | 2.995 | 3.718 | 3.969 | 4.518 | 4.499 | 4.860 | 2.767 |
| 8500 | 1.002 | 2.713 | 4.557 | 2.915 | 3.937 | 4.145 | 4.461 | 4.957 | 4.565 | 3.062 |
| 9000 | 1.004 | 2.872 | 4.656 | 3.256 | 3.820 | 4.021 | 4.435 | 4.684 | 4.718 | 2.961 |
| 9500 | 1.004 | 2.944 | 4.279 | 3.890 | 3.855 | 3.154 | 3.922 | 4.686 | 4.296 | 2.765 |
| 10000 | 0.995 | 2.933 | 4.686 | 4.059 | 3.736 | 4.214 | 3.742 | 4.724 | 4.401 | 2.932 |

Table 6: A table representing the speedup of the dynamic Mandelbrot computation run averages.

Figure 8: A graph representing the efficiency of the static Mandelbrot computation run averages.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Static Job Allocation Efficiency | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 0.462 | 0.340 | 0.388 | 0.380 | 0.397 | 0.377 | 0.360 | 0.347 | 0.056 | 0.324 |
| 1000 | 0.561 | 0.577 | 0.532 | 0.362 | 0.448 | 0.183 | 0.162 | 0.158 | 0.131 | 0.083 |
| 1500 | 0.488 | 0.549 | 0.646 | 0.430 | 0.465 | 0.340 | 0.315 | 0.239 | 0.254 | 0.118 |
| 2000 | 0.501 | 0.600 | 0.664 | 0.542 | 0.585 | 0.462 | 0.424 | 0.289 | 0.351 | 0.332 |
| 2500 | 0.497 | 0.675 | 0.718 | 0.607 | 0.527 | 0.425 | 0.357 | 0.494 | 0.533 | 0.303 |
| 3000 | 0.499 | 0.711 | 0.764 | 0.735 | 0.417 | 0.484 | 0.584 | 0.546 | 0.561 | 0.386 |
| 3500 | 0.497 | 0.716 | 0.770 | 0.739 | 0.658 | 0.660 | 0.659 | 0.577 | 0.398 | 0.427 |
| 4000 | 0.497 | 0.725 | 0.784 | 0.669 | 0.744 | 0.641 | 0.553 | 0.622 | 0.658 | 0.586 |
| 4500 | 0.502 | 0.728 | 0.783 | 0.775 | 0.681 | 0.718 | 0.581 | 0.687 | 0.601 | 0.694 |
| 5000 | 0.504 | 0.737 | 0.813 | 0.728 | 0.745 | 0.730 | 0.617 | 0.667 | 0.614 | 0.701 |
| 5500 | 0.500 | 0.735 | 0.795 | 0.810 | 0.648 | 0.729 | 0.707 | 0.718 | 0.702 | 0.709 |
| 6000 | 0.502 | 0.732 | 0.809 | 0.753 | 0.741 | 0.731 | 0.747 | 0.748 | 0.712 | 0.755 |
| 6500 | 0.501 | 0.738 | 0.814 | 0.799 | 0.732 | 0.769 | 0.733 | 0.762 | 0.687 | 0.704 |
| 7000 | 0.500 | 0.739 | 0.807 | 0.785 | 0.745 | 0.752 | 0.719 | 0.679 | 0.753 | 0.736 |
| 7500 | 0.500 | 0.740 | 0.821 | 0.809 | 0.731 | 0.758 | 0.739 | 0.734 | 0.740 | 0.775 |
| 8000 | 0.501 | 0.740 | 0.822 | 0.775 | 0.728 | 0.740 | 0.755 | 0.752 | 0.726 | 0.816 |
| 8500 | 0.500 | 0.744 | 0.825 | 0.798 | 0.731 | 0.764 | 0.761 | 0.768 | 0.731 | 0.797 |
| 9000 | 0.504 | 0.744 | 0.810 | 0.808 | 0.757 | 0.746 | 0.771 | 0.663 | 0.750 | 0.715 |
| 9500 | 0.502 | 0.744 | 0.823 | 0.822 | 0.734 | 0.733 | 0.720 | 0.762 | 0.760 | 0.732 |
| 10000 | 0.499 | 0.739 | 0.818 | 0.806 | 0.762 | 0.765 | 0.788 | 0.743 | 0.756 | 0.744 |

Table 7: A table representing the efficiency of the static Mandelbrot computation run averages.

Figure 9: A graph representing the efficiency of the average dynamic Mandelbrot computation runs.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dynamic Job Allocation Efficiency | | | | | | | | | | |
|  | Number of Slaves | | | | | | | | | |
| Image Size (Pixels) | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 |
| 500 | 0.501 | 0.495 | 0.242 | 0.081 | 0.302 | 0.145 | 0.229 | 0.344 | 0.103 | 0.116 |
| 1000 | 0.548 | 0.661 | 0.305 | 0.271 | 0.546 | 0.306 | 0.331 | 0.258 | 0.106 | 0.091 |
| 1500 | 0.503 | 0.659 | 0.407 | 0.462 | 0.462 | 0.474 | 0.414 | 0.320 | 0.186 | 0.137 |
| 2000 | 0.499 | 0.702 | 0.525 | 0.628 | 0.507 | 0.564 | 0.514 | 0.469 | 0.259 | 0.154 |
| 2500 | 0.501 | 0.720 | 0.385 | 0.634 | 0.692 | 0.661 | 0.562 | 0.540 | 0.486 | 0.138 |
| 3000 | 0.506 | 0.725 | 0.545 | 0.666 | 0.599 | 0.556 | 0.433 | 0.503 | 0.390 | 0.111 |
| 3500 | 0.500 | 0.740 | 0.571 | 0.621 | 0.393 | 0.429 | 0.454 | 0.243 | 0.445 | 0.119 |
| 4000 | 0.495 | 0.745 | 0.618 | 0.611 | 0.325 | 0.400 | 0.321 | 0.330 | 0.293 | 0.112 |
| 4500 | 0.504 | 0.760 | 0.712 | 0.626 | 0.295 | 0.310 | 0.302 | 0.352 | 0.271 | 0.120 |
| 5000 | 0.502 | 0.752 | 0.745 | 0.589 | 0.316 | 0.348 | 0.263 | 0.272 | 0.224 | 0.143 |
| 5500 | 0.497 | 0.754 | 0.694 | 0.435 | 0.304 | 0.288 | 0.270 | 0.303 | 0.314 | 0.128 |
| 6000 | 0.503 | 0.764 | 0.619 | 0.464 | 0.311 | 0.296 | 0.259 | 0.257 | 0.253 | 0.133 |
| 6500 | 0.498 | 0.757 | 0.769 | 0.430 | 0.339 | 0.331 | 0.278 | 0.275 | 0.211 | 0.120 |
| 7000 | 0.502 | 0.755 | 0.763 | 0.463 | 0.325 | 0.331 | 0.269 | 0.261 | 0.250 | 0.115 |
| 7500 | 0.500 | 0.621 | 0.748 | 0.464 | 0.386 | 0.284 | 0.291 | 0.279 | 0.233 | 0.142 |
| 8000 | 0.499 | 0.704 | 0.779 | 0.374 | 0.372 | 0.331 | 0.323 | 0.281 | 0.270 | 0.138 |
| 8500 | 0.501 | 0.678 | 0.760 | 0.364 | 0.394 | 0.345 | 0.319 | 0.310 | 0.254 | 0.153 |
| 9000 | 0.502 | 0.718 | 0.776 | 0.407 | 0.382 | 0.335 | 0.317 | 0.293 | 0.262 | 0.148 |
| 9500 | 0.502 | 0.736 | 0.713 | 0.486 | 0.385 | 0.263 | 0.280 | 0.293 | 0.239 | 0.138 |
| 10000 | 0.498 | 0.733 | 0.781 | 0.507 | 0.374 | 0.351 | 0.267 | 0.295 | 0.244 | 0.147 |

Table 8: A table representing the efficiency of the average dynamic Mandelbrot computation runs.

**Issues**

The primary issues with this experiment were associated with the grid. Not only was the grid experiencing failures during the time this assignment, but the very nature of the grid led to inconsistencies in the data. This led to inconsistent job performance due to the varying levels of traffic on the grid, and the variety of hardware included in the network also led to inconsistent results. Quite frankly, the grid made this assignment awful, considering the poor performance of the grid that would not grant enough TCP connections or speedy message passing, bad etiquette from others, and long waiting periods (I had a job wait from 11:30am to 2:00 am to begin working at one point). Devyani and Dr. Harris should not take this personally, this section is just for identifying any hindrances to success on the project.

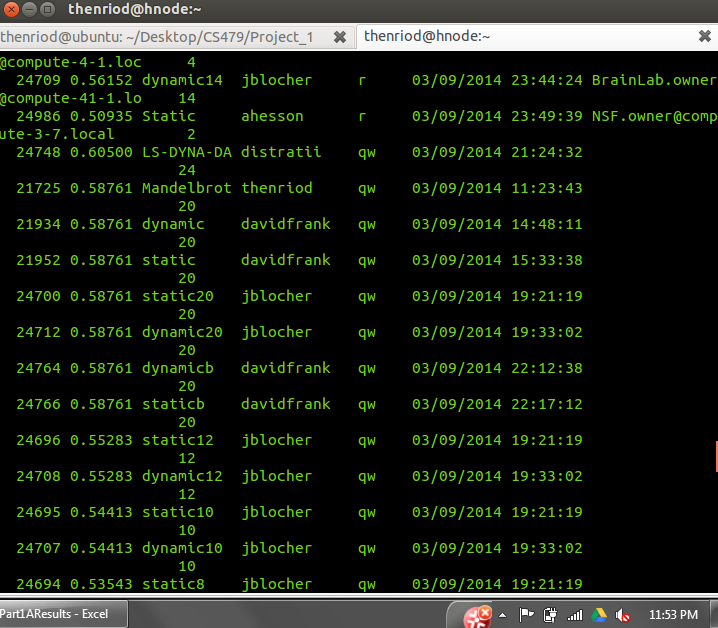


Figure 10: An example illustrating the difficulty of working on the grid in its current state. Note the drastic amount of time a job spent in the queue before getting to run. The terminal times are in 24:00 time, so this figure displays a >12 hour wait time.