**ECS 158 Project 3**

Sean Malloy, 998853013

Joseph,

ECS 158, Joël Porquet

11/18/2018

**Program 1: Matrix-multiplication with message passing**

We decided to implement matrix multiplication in multiply\_mpi by using the ikj matrix multiplication from Project 1. The main function handles input validation and initializes the MPI environment. We separated the work by a master-slave working paradigm with com0 being the master and the rest of the created processor coms being the slaves. Initially the master will allocate and initialize the matrices A, B and C. The master will then send the relevant chunk sizes, parts of the matrix A and also send B via MPI\_Send and MPI\_Bcast. We sent the whole B matrix since our Matrix Multiplications required it to work correctly, and we utilized MPI\_Bcast to deliver the data since this seemed to be faster than using the MPI\_Send method in certain situations, saving as much as 40 seconds in some runtimes. After the master sends the data, the slaves will receive these parameters via MPI\_Recv/MPI\_Bcast, and then continue to start working on their own pieces. The slaves will calculate the relevant parts of the C matrix independently. They will all use the multiply\_mpi function to calculate a piece of C and send the calculated piece back to the master via MPI\_Send. The master will be listening for these messages using MPI\_Recv and an offset variable will keep track of which parts of the C matrix are being received to rebuild the C matrix.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Locally |  |  |  |  |  |  |
| N | 8 Processors | 4 Processors | 2 Processors | Serial | OMP | Pthread |
| 500 | 0.046298 | 0.039501 | 0.034915 | 0.076914 | 0.042465 | 0.020875 |
|  | 0.045176 | 0.039078 | 0.034759 | 0.080405 | 0.019945 | 0.022151 |
|  | 0.045816 | 0.04053 | 0.034579 | 0.072048 | 0.029095 | 0.021396 |
| Average | 0.045763333 | 0.039703 | 0.034751 | 0.076455667 | 0.030501667 | 0.021474 |
| 1000 | 0.416857 | 0.394218 | 0.422085 | 0.577559 | 0.14836 | 0.14603 |
|  | 0.39572 | 0.42886 | 0.420755 | 0.578746 | 0.220676 | 0.143488 |
|  | 0.450489 | 0.430156 | 0.422429 | 0.587785 | 0.238844 | 0.141652 |
| Average | 0.421022 | 0.41774467 | 0.421756333 | 0.581363333 | 0.202626667 | 0.143723333 |
| 2000 | 3.136299 | 3.476246 | 3.338978 | 5.477773 | 1.310048 | 1.159077 |
|  | 3.252358 | 3.309672 | 3.338753 | 5.469149 | 1.564502 | 1.158264 |
|  | 3.098507 | 3.48029 | 3.338762 | 5.485127 | 1.591594 | 1.173145 |
| Average | 3.162388 | 3.42206933 | 3.338831 | 5.477349667 | 1.488714667 | 1.163495333 |
| 3000 | 10.477089 | 11.670172 | 11.200275 | 19.187115 | 3.86367 | 3.940683 |
|  | 10.70196 | 11.392818 | 11.164662 | 19.175987 | 3.997528 | 3.891222 |
|  | 10.463212 | 11.585681 | 11.204879 | 19.192328 | 4.011433 | 4.026503 |
| Average | 10.54742033 | 11.549557 | 11.18993867 | 19.18514333 | 3.957543667 | 3.952802667 |

For the local times for this implementation, we can see that if we increased the number of processors for MPI then it would be slower for smaller matrix sizes due to overhead compared to runs with less processors but will become much faster in larger matrix sizes since it can parallelize the data better to get it done much faster. Compared to OMP and Pthread MPI seems to be a bit slower, but that’s because of the way that MPI must parallelize its data creating more overhead for its functions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cluster |  |  |  |  |  |
| N | 16 Processors | 8 Processors | 4 Processors | 2 Processors | Serial |
| 500 | 0.460313 | 0.766185 | 0.033531 | 0.037147 | 0.062168 |
|  | 0.462432 | 0.790398 | 0.027087 | 0.036165 | 0.059812 |
|  | 0.46448 | 0.78993 | 0.036316 | 0.036225 | 0.062659 |
| Average | 0.462408333 | 0.782171 | 0.032311333 | 0.036512333 | 0.061546333 |
| 1000 | 1.948763 | 1.565718 | 0.346518 | 0.348843 | 0.588272 |
|  | 1.946731 | 1.573371 | 0.361926 | 0.376295 | 0.590393 |
|  | 1.939037 | 1.578041 | 0.367589 | 0.394079 | 0.587093 |
| Average | 1.944843667 | 1.57237667 | 0.358677667 | 0.373072333 | 0.588586 |
| 2000 | 8.973455 | 6.99989 | 2.840205 | 3.065135 | 4.834275 |
|  | 8.948229 | 6.988381 | 2.867304 | 3.06897 | 4.827524 |
|  | 8.97409 | 6.995889 | 2.852408 | 3.06989 | 4.818571 |
| Average | 8.965258 | 6.99472 | 2.853305667 | 3.067998333 | 4.82679 |
| 3000 | 21.055664 | 17.302096 | 9.535181 | 10.436245 | 16.811883 |
|  | 21.044934 | 17.272325 | 9.507568 | 10.448655 | 16.930845 |
|  | 20.999419 | 17.273446 | 9.485279 | 10.423371 | 16.871649 |
| Average | 21.033339 | 17.2826223 | 9.509342667 | 10.43609033 | 16.871459 |
| 5000 | 63.151831 | 56.784605 | 43.748199 | 47.881802 | 77.79539 |
|  | 62.639295 | 56.687771 | 43.608063 | 48.062257 | 78.055581 |
|  | 62.718172 | 56.86574 | 43.699882 | 47.966169 | 77.667931 |
| Average | 62.83643267 | 56.779372 | 43.68538133 | 47.970076 | 77.839634 |

For the timing of the MPI\_Matrix multiplication, when we parallelize the data over a cluster, then it seems that the larger number of processors carry more overhead, as it needs to wait to communicate with the other computers over the network, and thus fewer processors might be better in some situations. We can see that having it done serially is usually slower than doing it parallelized, unless if we have to communicate with multiple computers to release the data to.

**Program 2: Mandelbrot**

We use the same master-slave working paradigm as in Program 1 for Program 2, with com0 being the master and coms being the slaves. The main function again handles the input validation and initialization of the MPI environment. The master starts out by calculating the relevant chunk sizes and starting indexes and sends the parameters that the slaves will be using with a combination of MPI\_Send and MPI\_Broadcast (Bcast). (chunk\_size, start, xcenter, ycenter, cutoff, increment). The slaves will then receive all the parameters via a combination of MPI\_Recv and MPI\_Broadcast and start working on the chunk they have received.

The function compute\_mandelbrot will map the complex plane with the 1024x1024 matrix. We iterate over the 1024x1024 matrix and align the index of the image of the mandelbrot set with the matrix. In order to map the correct image, we had to start iterating from the bottom left corner of the image. We calculate the x distance from the center to the left side of the image using xcenter and the increment and since we know that it is being mapped to a 1024x1024 matrix the distance would equate to 512\*increment. We do the same for the y distance and with the new x and y coordinates we have found the bottom left corner. Using the start and chunk sizes we are able to increment our x’s and y’s independently in order to calculate whether values are in mandelbrot set or not. znext = zcurr + zcurr \* C with C = xvalue + yvalue \* I and |znext| > 2. Once the mandelbrot set is calculated for the chunk the chunk is returned back to the master. The master rebuilds the 1024x1024 matrix from the slaves using MPI\_Recv. The matrix is transposed and written to a file to give the output mandelbrot pgm file.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MandelBrot |  |  |  |  |  |  |
| Cutoff | 32 Processors | 16 Processors | 8 Processors | 4 Processors | 2 Processors | Serial |
| 100 | 0.548588 | 0.372809 | 0.220856 | 0.150223 | 0.173559 | 0.18545 |
|  | 0.7558 | 0.382379 | 0.222744 | 0.145613 | 0.176846 | 0.184389 |
|  | 0.700993 | 0.369012 | 0.218491 | 0.167863 | 0.175918 | 0.181043 |
| Average | 0.668460333 | 0.37473333 | 0.220697 | 0.154566333 | 0.175441 | 0.183627333 |
| 300 | 0.686276 | 0.426595 | 0.361097 | 0.407893 | 0.450675 | 0.461274 |
|  | 0.749883 | 0.400628 | 0.32376 | 0.409385 | 0.457917 | 0.458505 |
|  | 0.836359 | 0.369147 | 0.34272 | 0.386961 | 0.450339 | 0.452632 |
| Average | 0.757506 | 0.39879 | 0.342525667 | 0.401413 | 0.452977 | 0.457470333 |
| 500 | 1.088704 | 0.547712 | 0.490293 | 0.658137 | 0.7263 | 0.742486 |
|  | 0.982334 | 0.485668 | 0.525751 | 0.654951 | 0.725829 | 0.728372 |
|  | 0.937291 | 0.511387 | 0.499431 | 0.672994 | 0.724399 | 0.7282 |
| Average | 1.002776333 | 0.51492233 | 0.505158333 | 0.662027333 | 0.725509333 | 0.733019333 |
| 800 | 1.282963 | 0.758324 | 0.769739 | 1.025645 | 1.134734 | 1.133429 |
|  | 1.473108 | 0.756142 | 0.736083 | 1.032585 | 1.128297 | 1.135651 |
|  | 1.409152 | 0.747725 | 0.739019 | 1.029471 | 1.147957 | 1.135848 |
| Average | 1.388407667 | 0.75406367 | 0.748280333 | 1.029233667 | 1.136996 | 1.134976 |

As for the times, we had utilized a xcenter = -1, y center= -1, and zoom = 0.1 and differing cutoff values since that seemed the most optimal way to check the difference in speedup. From what we saw, if we had too many processors, then the speed will slow down due to overhead, but with about 8 processors we see the greatest speedup except for a small cutoff point. We also see Serial being pretty decent for a small cutoff point, but get slower compared to our 8 processor implementation with larger cutoffs.