HW3 Report CS 1645

Nicholas Zullo (njz12)

Problem 1: Matrix Multiplication

Introduction:

Matrix Multiplication is the task of multiplying 2 matrices together, into a resulting matrix. The dimensions of the matrices being multiplied must be of the form [b,a]\*[a,c] = [b,c]. In this problem, we are multiplying 3 square matrices, A,B, and C, so the dimensions will always match. To perform the multiplication, the index [i, j] of the resulting matrix will be the dot product of row i from matrix A and column j from matrix B. place the result of the dot product into index [i,j]. We perform this dot product for each row and column. To implement this serially, we don’t need to finish multiplying A and B before starting to compute the second multiplication. We only need the finished row from A\*B that will be used in the second multiplication. This allows us to save time by not iterating on the rows more than once.

Solution:

This was the most challenging to implement with MPI out of the 3 problems for me. The first thing that needs changed is the matrices themselves. In order to send the data with MPI, we need to change the declaration from 2D arrays to an int pointer. These pointers will have space allocated for NROW\*NCOL\*sizeof(int). This will be enough space to hold all elements of the matrix. Whenever we access the matrix now, we must use pointer arithmetic to reach the same values that [i][j] syntax would reach. For the row, we use i\*NCOL. This moves the pointer past i rows of data in memory. Next we need the column, which is simply j. The total value is then (i\*NCOL)+j. We will index the matrices using this syntax, where i is the current row and j is the current column. After initializing the matrices, we launch MPI with MPI\_Init(). The next thing that needs done is sending the data to each process. We will first declare new local copies of the matrices that will be filled next. We will use MPI\_Scatterv() to send the data. We need to use Scatterv to vary the amount of data that will be sent to the last process in case the number of processes does not evenly divide the data. When calculating the amount of data for the last process, we must use a specific formula, ((NROW/size) + (NROW % size)) \* NROW, where size is the number of processes. We first calculate the number of rows that each process will get, then add the remaining rows. The last operation, \* NROW, converts the row numbers into a 1 dimensional form for Scatterv. We define “chunk” to be the 1 dimensional amount of data given to each process. This means that the displacement for Scatterv will be an array of chunk\*i. The count for each process will be an array of chunk at each index, with the last index equal to the special case formula for the last process. We are now ready to send the data to each process. After receiving the data, each process calculates the local matrix. We then Gatherv() to process 0, using the same displacement and count values as before. Process 0 then confirms the validity of the final matrix, and we end by calling MPI\_Finalize().

Experiments:

|  |  |  |
| --- | --- | --- |
| NROW | # Processes | Runtime (ms) |
| 1024 | 1 | 12714.7 |
| 1024 | 4 | 1481.2 |
| 1024 | 16 | 1171.45 |
| 1024 | 32 | 1851.1 |
| 2048 | 1 | 128664 |
| 2048 | 4 | 13806.7 |
| 2048 | 16 | 7064.22 |
| 2048 | 32 | 8198.08 |

Conclusion:

There is a clear speed up and efficiency gain by utilizing MPI. In the best case, using 16 processes, we see a speed up of 10.85, and an efficiency of .678 with an NROW of 1024. This is a large improvement over the serial execution. When we increase the problem size to 2048, we see a speed up of 18.21 and an efficiency of 1.13 using 16 processes. As the problem size increases, we see greater benefits from using parallel programming. Using this method to parallelize came at a high engineering cost however. It took a long time to write the code to utilize MPI. However, we still see large speedups, and if this problem is executed many times, it is worth the effort to utilize MPI.

Problem 2: Mandelbrot Set

Introduction:

The Mandelbrot set is a mathematical series produced by: the set of complex numbers, c, where fc(z) = z2+c does not diverge when z is iterated from 0 to N where N is the stopping point. For example, at c=1, the sequence produces 0, 1, 2, 5, 26, which diverges toward infinity, so c=1 does not belong to the Mandelbrot set. However, for c=-1, the sequence is 0, -1, 0, -1, which does not diverge, so c=-1 belongs to the Mandelbrot set. The serial version of this code works to fill a matrix of pixels with the value of the color of that pixel. This number is found by iterating over each row and each column of the matrix, and calculating values from z=0 to N. The number of iterations before a mathematical condition is broken, or until N is exceeded, will be placed into the matrix as the color of pixel[i-1][j-1]. The previous pixel is used because we need to start iterating at 1 instead of 0 for the calculations. After filling the matrix, we find the total sum over all rows and columns and divide each cell by total/(N\*N). This normalizes the numbers in the matrix. We use N\*N because that is the number of pixels, and the average is total sum/count.

Solution:

This was another challenging problem to implement with MPI. We will follow a similar methodology as matrix multiplication to complete this problem. We first flatten the pixels array to be malloc(N \* N \*sizeof(int)). Next we will launch MPI. We calculate the chunk in this problem to be N / size. This is the number of iterations for each process to calculate. To convert this to a 1D value for data sending, we multiply by N again to account for the 2nd dimension of the array. We have no data to send initially, so we can begin the loop. The first loop begins at 0 and ends after the number of iterations for that process. We must change 1 thing about the algorithm in order to create the proper matrix. Calculating x0 initially needs to use the effective value of i, meaning the position in the calculation that we would be in if doing this serially. To calculate this, we create a variable start\_i. This will be the number of iterations each process completes. We multiply this by the rank of the process to reach the starting position of the calculation for this process. We add i to this value to properly calculate the value of x0 for each iteration. We will also track the sum calculated by each process now so we do not need to go back and find it later. We store the calculated count in the local array of pixels. After finishing the calculations, we perform an Allreduce() to calculate the total sum and give that to each process. We now normalize the local pixels array using the average calculated with the total sum. After normalizing the array, we are ready to gather the pixels array. Again, we need to Gatherv() to account for uneven data. The count and displacements use the same methodology as matrix multiplication. After gathering the data, we use process 0 to finish the program, and then MPI\_Finalize() from all processes.

Experiments:

|  |  |  |
| --- | --- | --- |
| N | # Processes | Runtime (ms) |
| 10000 | 1 | 6528.13 |
| 10000 | 4 | 3589.56 |
| 10000 | 16 | 2995.53 |
| 10000 | 256 | 3720.61 |
| 20000 | 1 | 25198.9 |
| 20000 | 4 | 13123.8 |
| 20000 | 16 | 11198.8 |
| 20000 | 256 | 12204.6 |

Conclusion:

We see a significant difference in the run time between the serial and MPI versions of the code. With 16 processes, we see a speed up of 2.17, and an efficiency of .13. While the efficiency number isn’t as close to 1 as we’d like, decreasing the runtime by half is still a good result. We also see a speed up of close to double with 4 processes, reaching 1.81, and the efficiency number now seems much better at .455. When we increase the problem size, we see similar results. The speedup with 16 processes and an N of 20000 is 2.25 and the efficiency is again not good at .14. Using 4 processes gives us a better number for the efficiency while still having similar numbers for the speedup, with a speedup of 1.92 and efficiency of .48. We should use MPI to increase the speed of this problem over serial execution. The engineering effort was again high, but can still be offset by the time savings if this problem needs to be ran a large amount of time.

Problem 3: Trapezoidal Approximation

Introduction:

The trapezoidal rule is a way to approximate the integral of a function over a given range. The area under the curve is split into NSTEPS trapezoidal sections. The area of each trapezoid is calculated and added together to create the total area under the curve in the given range. To serially solve this problem, simply loop over all NSTEPS and compute the area of the trapezoid, then add it to the total area.

Solution:

This problem was the easiest of the 3 to implement in MPI. This is because there is a minimal amount of work needed to collect data, and no work needs done to distribute the data. We follow the same initialization steps as the previous 2 problems, but no 2D arrays are present this time. Each process will calculate “chunk” number of iterations. We must change 1 thing about the algorithm, similar to what we needed to do in Mandelbrot. We must again track the effective position of each process in the calculation. This is because we need the process in the middle to calculate the middle portion of the integral. We track this in the same way as before, by using chunk\*rank + i to create the effective i position. We will store the calculated area in a local variable. After each process calculates its local area, we will perform a reduction to gather the sum to process 0. Process 0 then finishes the program, and each one calls MPI\_Finalize().

Experiments:

|  |  |  |
| --- | --- | --- |
| N | # Processes | Runtime (ms) |
| 2000 | 1 | 2.925 |
| 2000 | 4 | 16.039 |
| 2000 | 16 | 17.87 |
| 20000 | 1 | 26.299 |
| 20000 | 4 | 16.096 |
| 20000 | 16 | 15.782 |

Conclusion:

Here we see a surprising result at first. The problem size is not large enough to offset the extra time MPI uses to run on distributed systems. We see a very significant slowdown in runtime, not a speedup. This is a case where we need to increase the problem size in order to see better performance from MPI. When we increase from N=2000 to N=20000, we see MPI is now able to achieve better performance than the serial execution. We see a speed up of 1.63 and efficiency of .41 with 4 processes, and speedup of 1.66 and efficiency of .104 with 16 processes. We again reach the point where adding more processes does not help to increase the runtime by a significant margin. However, we should still use MPI to increase the speed of the execution if the problem size is large enough. The engineering cost was simple for this problem, so it will not take long for the time investment to parallelize the code to be passed by the time savings.