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:

To parallelize the code using OMP, we only need a single pragma, #pragma omp parallel for  num\_threads(NUM\_THREADS) schedule(guided, 100) default(none) shared(tempMatrix, matrixA, matrixB, matrixC, outputMatrix) private(i,j,k). Let’s break this down. “#pragma omp parallel for” tells OMP to create threads that will be used to parallelize a for loop. Next, we define the number of threads to be equal to the value contained in the Macro NUM\_THREADS. Next, one of the most important parts is how should the threads divide the work. We will experiment with different methods of dividing the work, and different number of threads, in the next section. We will compare with static, meaning each thread gets an even number of loop iterations as defined by the chunk size, guided, which does automatic load balancing based on the chunk size, and dynamic, which provides threads with work as needed. After defining the work load, we signal what purpose each variable will have, whether it will be shared by all threads or private to each thread.

Now that OMP is setup, the reason it works is simple. Since there are no dependencies, we can divide the work into chunks between threads without any other synchronization methods. We will now look at ways to divide the workload to get the biggest speedup possible.

Experiments:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NROW | N Threads | Schedule | Chunk Size | Runtime (ms) |
| 1024 | 1 | N/A | N/A | 12707.1 |
| 1024 | 8 | Guided | 100 | 2726.4 |
| 1024 | 8 | Guided | 200 | 3211.42 |
| 1024 | 8 | Guided | 50 | 2161.34 |
| 1024 | 8 | Static | 50 | 1953.15 |
| 1024 | 8 | Static | 100 | 2607.48 |
| 1024 | 32 | Static | 50 | 971.785 |
| 1024 | 32 | Static | 100 | 1653.27 |
| 1024 | 32 | Guided | 100 | 1614.89 |
| 1024 | 32 | Guided | 50 | 1059.64 |
| 1024 | 32 | Dynamic | 50 | 1055.22 |
| 2048 | 1 | N/A | N/A | 129645 |
| 2048 | 8 | Guided | 50 | 16433.6 |
| 2048 | 32 | Static | 50 | 8322.89 |

The speed of the parallelization depends greatly on parameters that need to be set by the programmer. We see that any parallelization creates a speed up when compared to the serial. However, there is room to be improved compared to the first row attempt of parameters. The Initial choices of 8 threads, guided, and 100 chunk size creates a speed up of 4.66. The best parameters, 32 threads, static, 50 chunk size, creates a speed up of 13.07. The same is true when we increase the amount of data. We see a greater speedup when using these ideal parameters.

Conclusion:

Parallelizing matrix multiplication allows for a huge speedup. This comes at a low engineering cost as well. The code to parallelize this is not hard to write. Utilizing OMP allows us to parallelize with only 1 extra line of code. If you need to perform a large amount of matrix multiplications, then the time taken to parallelize it is well invested compared to the speedups achieved. The serial version of matrix multiplication should only be used in situations where it is impossible to run multithreaded code. Otherwise, the speedup is too great to not utilize parallelism.

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:

We will parallelize this code in 3 chunks. The first chunk is filling the pixels matrix, the next is calculating the average, and the last is applying the normalization to the pixels matrix. We need a barrier between all 3 of these chunks. The average cannot be calculated until pixels is filled, and the normalization cannot be applied until after the average is calculated. This is a true dependency between chunk 1 and chunk 2, and a true dependency between chunk 2 and chunk 3. Each chunk can be parallelized individually in similar ways, however.

Chunk 1 and 3 both use the same parallelization methodology as Matrix Multiplication. Divide the work of the loops into chunks. There are no dependencies inside chunk 1 or 3 that require any other synchronization techniques such as locks. Chunk 2 can be parallelized using a reduction. Calculating the sum over a list of values is a common parallel problem and is well solved by a reduction. Reductions automatically divide the work of the addition to utilize as many processors as possible.

Experiments:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | # Threads | Schedule | Chunk Size | Runtime (ms) |
| 10000 | 1 | N/A | N/A | 6463.36 |
| 10000 | 8 | Static | 100 | 1146.58 |
| 10000 | 8 | Static | 50 | 1132.46 |
| 10000 | 256 | Static | 100 | 494.524 |
| 10000 | 256 | Static | 50 | 404.03 |
| 10000 | 256 | Guided | 50 | 411.289 |
| 10000 | 512 | Guided | 25 | 443.382 |
| 10000 | 256 | Guided | 10 | 345.978 |
| 10000 | 256 | Guided | 5 | 364.832 |
| 10000 | 256 | Static | 10 | 370.907 |
| 20000 | 1 | N/A | N/A | 25915 |
| 20000 | 256 | Static | 20 | 1315.75 |
| 20000 | 256 | Guided | 20 | 1269.75 |

The parallel code clearly preforms way better than the serial version. With the first attempt parameters of 8 threads, static, 100 chunk, we see a speed up of 5.63. this is a large speedup, and the runtime is much quicker already. However, we can still optimize more. With the ideal parameters of 256 threads, guided, 10 starting chunk size, we reach a speedup of 18.68. This is a huge speedup. We see a similar result when increasing the data size from 10000 to 20000 as well. The ideal parameters of 256 threads, guided, and a starting chunk size of 20 result in a speed up of 20.41, even more than with N=10000.

Conclusion:

As long as the system can support multithreading, 256 threads should be used to maximize performance. Even when the problem size increases, 256 threads still handle the problem the best. Adding more threads with a larger problem size did not create better performance than with a smaller problem size. The engineering cost of creating the parallel solution again is minimal compared to the speedup. We need only 3 extra lines to turn the serial code into parallel using OMP. If Mandelbrot needs to be calculated frequently, it is worth it to use the time to create the parallel version.

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:

The parallel code of trapezoidal approximation is very similar to matrix multiplication. We have no dependencies, so no synchronization locks are needed. We can simply divide the work evenly between the threads and allow them to calculate the area. However, unlike matrix multiplication, we have a reduction now. A reduction is used to calculate the area for each thread. Each thread must add its partial sums into area, and a reduction is the best way to calculate parallel sums. One line of OMP is all that is needed to parallelize this code, and it is the same as matrix multiplication but with reduction(+:area) added to the end.

Experiments:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NSTEPS | # Threads | Schedule | Chunk Size | Runtime (ms) |
| 2000 | 1 | N/A | N/A | 2.866 |
| 2000 | 8 | Guided | 100 | .722 |
| 2000 | 8 | Static | 100 | .816 |
| 2000 | 16 | Guided | 100 | .885 |
| 2000 | 16 | Guided | 50 | .83 |
| 2000 | 16 | Static | 100 | .865 |
| 4000 | 1 | N/A | N/A | 5.679 |
| 4000 | 8 | Static | 100 | 1.058 |
| 4000 | 8 | Guided | 100 | 1.325 |

We see again that parallel code beats the performance of serial. This time, our original guess for the parameters actually turned out to be the fastest, giving a speed up of 3.97 using 8 threads, guided, starting with a chunk of 100. The performance was very similar through all the parameters which was surprising. Other chunk size and thread counts were used but were not shown here because they had the same runtime. Regardless, a speed up of nearly 4 is very significant, and shows the benefits of parallelization.

Conclusion:

Parallelism here creates a faster solution than serial. It is very simple to code, requiring only 1 extra line. The only limitation of this solution is if the system cannot support multithreading, then serial should be used. Otherwise, as with the previous 2 problems, parallelism is the ideal solution.