HW2 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 this code, we first need to look for dependencies. In this problem, there are no dependencies to work around, so we can begin to parallelize right away. First, we will discuss the function that each thread will run. Each thread will begin by calculating the data partition that it will be working with. To do this, we will simply divide the number of rows by the number of threads. This will work to evenly disperse the data along every thread as long as the number of threads divides evenly into the number rows, i.e. number rows % number threads == 0. If it does not divide evenly, we must account for this with the last thread. We will add (number rows % number threads) to the number of rows given to the last thread. This allows for all the data to be evenly dispersed and all rows to be given to a thread. After assigning the data partition, we will begin to iterate through the partition. This section will be very similar to the serial code. We do not need to use a mutex to protect the matrix we are writing to because each thread will be writing to its own section of the matrix, and there will be no overlap. If there was overlap and the threads were writing to the same location, we would need to use a lock and determine which thread receives writing priority. Since all the threads write to the same matrix, we do not need to do any more work to collect the results. From the main function, we will create the matrices A B and C, then create the threads and begin execution of the thread function. After creating the threads, the main function will wait and join the threads together to complete the program.

Experiments:

The serial code completes in 10313.6 milliseconds with a matrix size of 1028. This is compared to the parallel code which completes at a peak of 547.292 milliseconds with a matrix size of 1028 and 32 threads. This is 18 times faster! Starting with 2 threads, the performance increases when the number of threads increases until we start to exceed 32 threads. After 32, the number of threads does not decrease runtime, and after 48 the runtime begins increasing.

|  |  |  |
| --- | --- | --- |
| Number of Threads | Size of matrix | Runtime (ms) |
| 1 | 1028 | 10397.5 |
| 8 | 1028 | 1132.73 |
| 32 | 1028 | 547.292 |
| 1 | 2048 | 129615.0 |
| 32 | 2048 | 4534.77 |
| 48 | 2048 | 5393.93 |

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. 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:

To parallelize this code, again we look for dependencies. When filling the matrix, there are no dependencies or race conditions to account for because each loop is writing to its own spot in the matrix. There is a dependency on the average though. We cannot continue to normalizing the matrix until after all threads have finished calculating the pixel values. This means we must use a barrier. We will come back to the barrier soon. To write the parallel code, we must write a function for each thread to call to perform its work in. At the start of this function, we must partition the data so that each thread receives an evenly spaced amount of data to work with. This is done by dividing N, the number of rows in the grid of pixels, by the number of threads. If N % number of threads == 0, then the data is evenly distributed between the threads, and no data is leftover. If it does not equal 0, then some data is left to be partitioned. We will add the remaining data to the last thread’s data. This means the data is almost all evenly distributed, with some imbalance in the last thread, but no data is leftover and missed by the calculations. Each thread will then fill the pixel grid using the same code as the serial version, but only on its partition of the data. The next thing that needs done is normalizing the matrix of pixels. To find the total sum, we will not iterate over the pixel grid again. Each thread will instead track the partial sum of its data partition. After finishing its pixel grid, each thread will add its partial sum to a global variable, total sum. Total sum must be protected with a mutex because it is a shared variable. After finding the sum, each thread will normalize its partition of the data. However, it cannot do this before each thread has finished calculating and the total sum is found. This is where we will use a barrier. After adding to total sum, and before beginning to normalize the data, each thread will wait at a barrier until all threads reach the barrier. Only now will the threads move on to normalizing the pixels. Normalizing the pixels is straightforward, as there are no dependencies or shared data. No thread will be writing to the same region of the pixels matrix so no mutex is needed to protect the data. Once the threads finish normalizing, we do not need to do any more work to collect the output because each thread writes to pixels and pixels is a global variable.

The main thread will be the same as it was for matrix multiplication, but instead of initializing a matrix of dummy values we will initialize a barrier. Then, we will create all the threads and have them call the thread calculate function. Afterwards, the main thread will wait until each thread is done and join them together.

Experiments:

|  |  |  |
| --- | --- | --- |
| Thread Count | N | Runtime (ms) |
| 1 | 10000 | 6492.7 |
| 8 | 10000 | 2513.23 |
| 32 | 10000 | 817.266 |
| 48 | 10000 | 612.003 |
| 64 | 10000 | 509.557 |
| 128 | 10000 | 392.033 |
| 256 | 10000 | 352.315 |
| 512 | 10000 | 383.899 |
| 1 | 20000 | 25222.5 |
| 32 | 20000 | 2535.83 |
| 256 | 20000 | 1197.22 |
| 512 | 20000 | 1197.74 |

It is clear that parallelizing this problem creates massive speedups again. For N=10000, there is again an 18x speedup, which is a huge gain (from thread count 1 to 256). With Mandelbrot, the speed up requires more threads compared to matrix multiplication. With matrix multiplication, performance peaked after 32 threads. However, with Mandelbrot, performance doesn’t start to degrade until after 256. This is a potential limitation, as some systems cannot support that many threads and you will be forced to use less, resulting in less of a speedup. There is no false sharing or other parallelism limitations here.

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. 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 version of this code is simpler than the Mandelbrot code. No barriers are needed this time, but a mutex for a shared variable will be used. We will start again with the thread calculate function. Still, the first thing each thread will do is to calculate the data partition size. This is done by dividing NSTEPS by number of threads, which will evenly distribute the data if NSTEPS % number of threads == 0. Again, we will add the remaining data (NSTEPS % number of threads) to the last thread so that no data is left out. Now that each thread has its data partition, we will calculate the area of the trapezoids in the partition. We will add the areas to a partial sum for each thread. Once the area of the partition has been calculated in a partial sum, we will add it to the global variable total sum. Total sum is a shared variable, so it must be protected with a mutex. Only 1 thread can update the total sum at time. After each thread adds its partial sum to the global sum, we are done with the work in the threads. Back in the main thread, the only work that needs to be done is creating the threads and joining them once they are done calculating the area.

Experiments:

|  |  |  |
| --- | --- | --- |
| Thread Count | NSTEPS | Runtime (ms) |
| 1 | 2000 | 2.909 |
| 8 | 2000 | 1.539 |
| 16 | 2000 | .994 |
| 32 | 2000 | 1.659 |
| 1 | 4000 | 5.66 |
| 16 | 4000 | 1.746 |
| 32 | 4000 | 1.609 |
| 48 | 4000 | 2.322 |

Once again, parallelism creates a faster solution. The speedup this time is not as drastic as it was with the previous 2 problems, but is still significant. From 1 thread to 16, the speedup is nearly 3x. When the problem size increases, there is some speed up from increasing the threads, but not much, and it doesn’t take long before increasing the number of threads no longer creates a performance gain. This is similar to matrix multiplication where the optimal number of threads is relatively small. Again, parallelism is beneficial to use here because a 3x speedup is significantly faster than serial. The code to create this solution is simple, and the little time that is taken to create it is well worth the benefit of running faster code.

Conclusion:

Parallelism here creates a faster solution than serial. 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.