CS 415 Parallel Computing

Matrix Multiplication

April 18, 2012

Erin Keith

**Introduction:**

This project focused on computing the product of two matrices. The program was given two matrices to multiply together by summing the product of the first matrix’s row elements and the second matrix column elements, to produce each element of the resulting matrix. A matrix with **n** rows by **p** columns multiplied by a matrix with **p** rows by **m** columns results in a matrix of **n** rows by **m** columns:

**Procedure:**

*Sequential Program*

The sequential program was very simply comprised of 3 nested for loops used to calculate the sum for each of the resulting elements.

*Part II: Parallel*

My approach to parallelizing the problem was very similar to my Mandelbrot implementation. After broadcasting one of the matrices, I allowed for a workpool approach to dole out each row of the remaining matrix. The worker process then calculated the sum of the products of elements and returned the resulting array to the master process. The master process was responsible for inserting the row into the appropriate spot in the result matrix.

**Analysis:**

Because the sequential program has a time complexity of O(n3), significant speedup was expected. The runtimes for the sequential version of the program are included in Table 1.

**Table 1: Sequential Runtimes for Matrix Multiplication**

|  |  |
| --- | --- |
| **Elements** | **Time** |
| **200** | 0.29 |
| **400** | 1.41 |
| **600** | 4.33 |
| **800** | 10.66 |
| **1000** | 23.85 |
| **1200** | 40.76 |
| **1400** | 67.16 |
| **1600** | 101.57 |
| **1800** | 132.23 |
| **2000** | 183.74 |

**Figure 1: Runtimes for Sequential Matrix Multiplication Implementation**

Initially, I experienced some superlinear speedup, as did some of my peers. Since my parallel implementation only had two for loops in the worker, I suggest that the parallel implementation far better computation times. As it turns out, I had been dynamically declaring the arrays in my sequential program but not in my parallel, which accounted for the slower sequential runtimes. Table 2 presents runtimes for different number of processors and matrix sizes, which are visualized in Figure 2.

The drawback of my implementation was the amount of communication I included. I did not implement the look up table for keeping track of rows being processed so I had significantly more communication. I ran my parallel program for a range of processors, but had a hard time getting consistent runtimes. I think most of the runtime problems had to do with network congestion and the inconsistency of running in the common queue.

**Table 2: Parallel Runtimes for Matrix Multiplication**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elements** | **10** | **11** | **12** | **13** |
| **200** | 5.03 | 5.89 | 5.25 | 3.99 |
| **400** | 5.86 | 7.42 | 5.93 | 5.2 |
| **600** | 4.58 | 4.94 | 4.61 | 5.58 |
| **800** | 5.04 | 5.82 | 6.03 | 7.67 |
| **1000** | 6.09 | 6.14 | 7.34 | 8.01 |
| **1200** | 7.97 | 7.27 | 9.62 | 8.36 |
| **1400** | 9.52 | 9.54 | 10.29 | 8.6 |
| **1600** | 11.61 | 13.13 | 11.69 | 9.21 |
| **1800** | 16.09 | 14.58 | 14.6 | 11.13 |
| **2000** | 19.21 | 18.64 | 17.59 | 14.89 |

**Figure 2: Runtimes for Parallel Matrix Multiplication Implementation**

I also noticed that there wasn’t much difference with increasing the processors. In fact it appeared that runtimes were increasing due to increased processors. This is most likely an effect of too much communication. The speedup and efficiency graphs shown in Figures 2 and 3 are bumpy, demonstrating the inconsistency of the grid.

**Table 3: Speedup for Matrix Multiplication**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elements** | **10** | **11** | **12** | **13** |
| **200** | 0.057654 | 0.239389 | 0.055238 | 0.072682 |
| **400** | 0.240614 | 0.190027 | 0.237774 | 0.271154 |
| **600** | 0.945415 | 2.157895 | 0.939262 | 0.775986 |
| **800** | 2.115079 | 1.831615 | 1.767828 | 1.389831 |
| **1000** | 3.916256 | 6.638436 | 3.249319 | 2.977528 |
| **1200** | 5.114178 | 5.606602 | 4.237006 | 4.875598 |
| **1400** | 7.054622 | 10.64675 | 6.526725 | 7.809302 |
| **1600** | 8.748493 | 7.73572 | 8.688623 | 11.02823 |
| **1800** | 8.218148 | 9.069273 | 9.056849 | 11.8805 |
| **2000** | 9.56481 | 9.857296 | 10.44571 | 12.33983 |

**Figure 3: Speedup for Matrix Multiplication Implementation**

**Table 4: Efficiency for Matrix Multiplication**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elements** | **10** | **11** | **12** | **13** |
| **200** | 0.576541 | 2.176262 | 0.460317 | 0.55909 |
| **400** | 2.406143 | 1.727518 | 1.98145 | 2.085799 |
| **600** | 9.454148 | 19.61722 | 7.827187 | 5.96912 |
| **800** | 21.15079 | 16.65105 | 14.7319 | 10.691 |
| **1000** | 39.16256 | 60.34942 | 27.07766 | 22.90406 |
| **1200** | 51.14178 | 50.96911 | 35.30839 | 37.5046 |
| **1400** | 70.54622 | 96.78864 | 54.38937 | 60.07156 |
| **1600** | 87.48493 | 70.32472 | 72.40519 | 84.83254 |
| **1800** | 82.18148 | 82.44794 | 75.47374 | 91.38849 |
| **2000** | 95.6481 | 89.61178 | 87.04756 | 94.92173 |

**Figure 4: Efficiency for Matrix Multiplication Implementation**

**Conclusion:**

This parallel implementation was very similar to that of Mandelbrot, with the addition of utilizing the MPI Broadcast function. I, along with the other students I spoke with, was pretty uncomfortable with efficiency greater than 100 percent, as we should have been. This was a good lesson in taking a closer look at differences between implementations.