**PA03: Matrix Multiplication**

**A Sequential Algorithm vs. a Statically Allocated Modified SUMMA**

**Terence Henriod**

**Dr. Fred Harris**

**CS615: Parallel Computing**

**Wednesday April 9, 2014**

**Introduction**

Tasks like matrix multiplication are easily parallelizable. They are also good practice to experiment with large message passing MPI routines such as broadcast, scatter, gather, and of course their variable versions as well. They also provide good opportunity to observe super-linear speedup due to the fact that working with contiguous portions of matrices might be cache-able, while handling entire matrices might foil some of the performance gains of caching.

**Theory**

*Matrix Multiplication*

Matrix Multiplication can be summarized as “row times column.” This means that each resulting element from a matrix multiplication is found by the summing of pairwise multiplications across the index row of the left side matrix and the index column of the right side of the matrix. Formally defined, each element of a matrix multiplication result matrix is found by:

|  |  |
| --- | --- |
|  | [1] |

where A must be a matrix and B must be a matrix. In order for any single resulting element of a matrix multiplication to be computed, all of row *i* of A and all of column *j* of B are required, hence the reason for devising various parallel matrix multiplication algorithms.

*SUMMA and Modified SUMMA*

The Scalable Universal Matrix Multiplication Algorithm in its most naïve form simply distributes a row and a column to each processor to compute one element of the result matrix. This incurs more message passing than would be ideal however. So SUMMA is often modified in ways that cut down on message passing, such as broadcasting blocks of rows and columns and then sending the resulting row(s) or column(s) back.

In the version that was used for this assignment, the entirety of both matrices are read from a file on each node, and then each node is then responsible for performing multiplications using a portion of matrix A and the entirety of matrix B. This results in a relatively straightforward algorithm that is easily managed and coded.

(Definitions of speedup and efficiency are listed for completeness, this is not new information)

*Speedup*

When computing things in parallel, it is advantageous to know how much faster the parallel algorithm/program runs compared to a sequential one. This is one metric used to evaluate the quality of a parallel algorithm, the *speedup factor*. The speedup factor represents how many times faster the parallel algorithm using *p* processors is compared to the sequential one. The speedup factor, *S(p)*, is computed as follows:

|  |  |
| --- | --- |
|  | [2] |

where *p* is the number of processors used in the parallel algorithm, *ts* and *tp* are the respective running times for sequential and parallel runs of the algorithms, and *f* is the fraction of the work in a program that must be run sequentially. The second and third expressions are known as Amdahl’s law. Ideally, the speedup factor will represent a number of factors , but this is often not the case due to various factors including non-parallelizable segments of a job or inter-process communication overhead. If *Sp* is small relative to *p*  or even approaches 1, then the parallel algorithm should either be improved or not used. Should *Sp* ever exceed *p*, this is known as *super-linear speedup*. A super-linear speedup factor is often the product of the use of a runtime that resulted from a sub-optimal sequential algorithm.

*Efficiency*

Parallel algorithms can also be measured in terms of how well the time to run a program is used by all of the processors. Efficiency measures how well the processors are used (what percentage of the runtime the processors spend working). Efficiency is found by:

|  |  |
| --- | --- |
|  | [3] |

where *ts*, *tp*, and *S(p)* are defined as in [2].

**Pseudo-Code**

It should be noted that the following pseudo-code should not be considered functional code. Functional code will be attached in the report. I have left out many details that would arise in actual code for the sake of brevity and actually describing the algorithms. Each set of pseudo-code assumes that the matrix data has already been read in to the program, so only significant computation and data gathering actions are presented.

*Sequential Pseudo-Code*

In a sequential program, the entire matrix multiplication procedure is done on one processor:

Matrix multiplyMatrices( Matrix A, Matrix B )

{

// precondition: A and B have matching numbers of rows to

// columns

running\_sum

matrix C // matrix C will have the same number of

// rows as A and columns as B

startTimer()

// using every row of A

for( i = 0; i < A.rows; i = i + 1 )

{

// using every row of B

for( j = 0; j < B.columns; j = j + 1 )

{

// reset the intermediate sum variable

running\_sum = 0;

// using every element of each row/column

for( k = 0; k < A.rows; k = k + 1 )

{

running­\_sum = running\_sum + A[i][k] \* B[k][j]

}

// store the result

C[i][j] = running\_sum

}

}

run\_time = stopTimer()

return C

}

*Parallel Psuedo-Code*

In this program, the only thing that really differentiates a master node from a slave is the fact that the master is doing all of the gathering of the data. Otherwise, all nodes compute what shares of work each node will do based on their rank in the MPI world and will perform their share of the multiplications. Each node reads in the matrix so there is no message passing required at the start. Job allocation was clearly static in nature.

Master:

Matrix masterMultiplyMatrices( Matrix A, Matrix B )

{

// precondition: A and B have matching numbers of rows to

// columns

total\_time

computation\_time

running\_sum

matrix C // matrix C will have the same number of

// rows as A and columns as B

Array starting\_rows

Array shares\_of\_rows

starting\_rows, shares\_of\_rows =

computeStartingRowsAndShares( A.rows )

startTotalTimeTimer()

startComputationTimer()

// using just a share of rows of A

for( i = starting\_rows[my\_rank];

i < starting\_rows[my\_rank] + shares\_of\_row[my\_rank];

i = i + 1 )

{

// using every row of B

for( j = 0; j < B.columns; j = j + 1 )

{

// reset the intermediate sum variable

running\_sum = 0;

// using every element of each row/column

for( k = 0; k < A.rows; k = k + 1 )

{

running­\_sum = running\_sum + A[i][k] \* B[k][j]

}

// store the result

C[i][j] = running\_sum

}

}

computation\_time = stopComputationTimer()

C = gatherComputationsFromAll( starting\_rows, shares\_of\_rows,

C )

total\_time = stopTotalTimeTimer()

return C, computation\_time, total\_time

}

Slave:

Matrix slaveMultiplyMatrices( Matrix A, Matrix B )

{

// precondition: A and B have matching numbers of rows to

// columns

Running\_sum

matrix C // matrix C will have the same number of

// rows as A and columns as B

Array starting\_rows

Array shares\_of\_rows

starting\_rows, shares\_of\_rows =

computeStartingRowsAndShares( A.rows )

// using just a share of rows of A

for( i = starting\_rows[my\_rank];

i < starting\_rows[my\_rank] + shares\_of\_row[my\_rank];

i = i + 1 )

{

// using every row of B

for( j = 0; j < B.columns; j = j + 1 )

{

// reset the intermediate sum variable

running\_sum = 0;

// using every element of each row/column

for( k = 0; k < A.rows; k = k + 1 )

{

running­\_sum = running\_sum + A[i][k] \* B[k][j]

}

// store the result

C[i][j] = running\_sum

}

}

computation\_time = stopComputationTimer()

sendComputationsToMaster( C, starting\_rows[my\_rank],

shares\_of\_rows[my\_rank] )

return NULL

}

For completeness:

computeStartingRowsAndShares( num\_rows )

{

minimum\_share

remainder

// compute the smallest share any node should get

minimum\_share = num\_rows / num\_nodes // integer division

remainder = num\_rows mod num\_nodes

// distribute any extra rows to the nodes with low ranks

for( i = 0; i < remainder; i = i + 1 )

{

starting\_rows[i] = i \* (minimum\_share + 1)

// ^^^ offset of all previous shares

shares\_of\_rows[i] = minimum\_share + 1

}

// distribute the remaining minimum shares

for( ; i < num\_nodes; i = i + 1 )

{

starting\_rows[i] = i \* minimum\_share + remainder

// ^^^ offset of all previous shares

shares\_of\_rows[i] = minimum\_share

}

return starting\_rows, shares\_of\_rows

}

**Output Data Summaries**

The following tables and figures are representative of the output of the trial runs of the matrix multiplication runs, listed by appropriate category. This is pretty boring and monotonous, but accurate, so if Devyani wanted to skip it, she could and still be confident that all the necessary items were here.

It should be noted that my matrices used floats instead of integers, so I suspect that my times may have taken longer than others who used integers because float math is more complicated than integer math.

It is interesting to note that the runtimes all got faster at matrix size 4000 for some unknown reason, but also that the speedups were far lower than might be expected, again, for unknown reasons.

I suspect that my results did not have very good speedup due to the fact that my algorithm allocated both matrices in their entirety and therefore was not as able to take advantage of things like caching. I also noticed that as more people began to use the grid, my performance suffered. It is very difficult to tell why there was such a lack of speedup, given that using similar static job allocation methods for the Mandelbrot project got me some of the best speedups in the class, and yet, here the best speedup observed is a mere 9 when using 14 processors. This was rather disappointing to see, especially since the observed message passing times were nearly non-existent compare to the computation times.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Computing Nodes Used in Experiment | | | | | | | |
| 0: | compute-3-7.local | 5: | compute-3-7.local | 10: | compute-3-0.local | 15: | compute-3-8.local |
| 1: | compute-3-7.local | 6: | compute-3-7.local | 11: | compute-3-0.local | 16: | compute-3-8.local |
| 2: | compute-3-7.local | 7: | compute-3-7.local | 12: | compute-3-0.local | 17: | compute-3-8.local |
| 3: | compute-3-7.local | 8: | compute-41-15.local | 13: | compute-3-0.local | 18: | compute-3-8.local |
| 4: | compute-3-7.local | 9: | compute-41-15.local | 14: | compute-3-8.local | 19: | compute-3-8.local |

Table 1: The processor nodes used for the matrix multiplication runs. If a run used *n* processors, then processors 0through *n* – 1 were used.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Average Sequential Matrix Multiplication Times | | | | | |
|  | Square Matrix Size | | | | | |
| Matrix Size | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Runtime (s) | 15.66 | 129.31 | 434.01 | 1265.43 | 2030.56 | 4341.46 |

Table 2: The sequential matrix multiplication average runtimes.

Figure 1: A graph of the average sequential runtimes.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Average Parallel Total Runtimes | | | | | |
|  |  | Square Matrix Size | | | | | |
|  |  | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Number of Processors | 2 | 7.36 | 63.40 | 208.37 | 631.24 | 960.09 | 2153.20 |
| 4 | 3.94 | 32.62 | 110.86 | 322.32 | 490.57 | 1071.34 |
| 6 | 2.75 | 22.13 | 69.97 | 216.08 | 323.22 | 743.65 |
| 8 | 2.39 | 17.24 | 60.11 | 163.75 | 256.43 | 577.02 |
| 10 | 4.96 | 19.76 | 67.22 | 386.03 | 310.37 | 1555.96 |
| 12 | 3.00 | 21.38 | 56.48 | 320.93 | 253.21 | 1092.37 |
| 14 | 2.51 | 15.50 | 47.38 | 275.28 | 466.93 | 2326.31 |
| 16 | 5.07 | 30.59 | 91.63 | 663.04 | 432.17 | 2104.19 |
| 18 | 4.54 | 25.97 | 82.13 | 592.22 | 358.56 | 1999.53 |
| 20 | 4.14 | 22.44 | 72.28 | 528.32 | 628.78 | 1818.65 |

Table 3: The total runtimes for the parallel matrix multiplication.

Figure 2: A graph of average total runtimes for the parallel version.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Average Parallel Computation Times | | | | | |
|  |  | Square Matrix Size | | | | | |
|  |  | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Number of Processors | 2 | 7.36 | 63.39 | 208.35 | 631.21 | 960.06 | 2153.15 |
| 4 | 3.94 | 32.61 | 110.85 | 322.29 | 490.53 | 1071.28 |
| 6 | 2.75 | 22.12 | 69.95 | 216.05 | 323.18 | 743.59 |
| 8 | 2.39 | 17.23 | 60.09 | 163.72 | 256.38 | 576.95 |
| 10 | 4.95 | 19.73 | 67.16 | 385.93 | 310.22 | 1555.70 |
| 12 | 2.98 | 21.34 | 56.37 | 320.79 | 252.97 | 1092.01 |
| 14 | 2.48 | 15.42 | 47.24 | 275.10 | 466.24 | 2325.73 |
| 16 | 5.05 | 30.52 | 91.47 | 662.79 | 431.81 | 2103.59 |
| 18 | 4.50 | 25.85 | 81.95 | 591.83 | 357.91 | 1998.88 |
| 20 | 3.84 | 22.03 | 71.90 | 527.68 | 341.75 | 1817.15 |

Table 4: The average computation times for the parallel matrix multiplication.

Figure 3: A graph of average computation times for the parallel version.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Average Parallel Message Passing Times | | | | | |
|  |  | Square Matrix Size | | | | | |
|  |  | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Number of Processors | 2 | 0.001 | 0.005 | 0.012 | 0.022 | 0.033 | 0.049 |
| 4 | 0.002 | 0.007 | 0.016 | 0.027 | 0.041 | 0.062 |
| 6 | 0.003 | 0.008 | 0.017 | 0.029 | 0.044 | 0.066 |
| 8 | 0.003 | 0.009 | 0.018 | 0.031 | 0.051 | 0.072 |
| 10 | 0.012 | 0.026 | 0.056 | 0.098 | 0.151 | 0.259 |
| 12 | 0.026 | 0.047 | 0.111 | 0.147 | 0.240 | 0.354 |
| 14 | 0.023 | 0.084 | 0.139 | 0.180 | 0.694 | 0.583 |
| 16 | 0.027 | 0.068 | 0.158 | 0.253 | 0.352 | 0.603 |
| 18 | 0.045 | 0.119 | 0.181 | 0.388 | 0.648 | 0.653 |
| 20 | 0.302 | 0.413 | 0.371 | 0.643 | 0.302 | 1.495 |

Table 5: The average message passing times for the parallel matrix multiplication.

Figure 4: A graph of average message passing times for the parallel version.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Average Parallel Speedup Factors | | | | | |
|  |  | Square Matrix Size | | | | | |
|  |  | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Number of Processors | 2 | 2.13 | 2.04 | 2.08 | 2.00 | 2.11 | 2.02 |
| 4 | 3.97 | 3.96 | 3.91 | 3.93 | 4.14 | 4.05 |
| 6 | 5.70 | 5.84 | 6.20 | 5.86 | 6.28 | 5.84 |
| 8 | 6.55 | 7.50 | 7.22 | 7.73 | 7.92 | 7.52 |
| 10 | 3.16 | 6.54 | 6.46 | 3.28 | 6.54 | 2.79 |
| 12 | 5.22 | 6.05 | 7.68 | 3.94 | 8.02 | 3.97 |
| 14 | 6.25 | 8.34 | 9.16 | 4.60 | 4.35 | 1.87 |
| 16 | 3.09 | 4.23 | 4.74 | 1.91 | 4.70 | 2.06 |
| 18 | 3.45 | 4.98 | 5.28 | 2.14 | 5.66 | 2.17 |
| 20 | 0.30 | 5.76 | 6.00 | 2.40 | 3.23 | 2.39 |

Table 6: The average speedups for the parallel matrix multiplication.

Figure 5: A graph of the average speedups for the parallel version.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Average Parallel Computation Efficiency | | | | | |
|  |  | Square Matrix Size | | | | | |
|  |  | 800 | 1600 | 2400 | 3200 | 4000 | 4800 |
| Number of Processors | 2 | 1.06 | 1.02 | 1.04 | 1.00 | 1.06 | 1.01 |
| 4 | 0.99 | 0.99 | 0.98 | 0.98 | 1.03 | 1.01 |
| 6 | 0.95 | 0.97 | 1.03 | 0.98 | 1.05 | 0.97 |
| 8 | 0.82 | 0.94 | 0.90 | 0.97 | 0.99 | 0.94 |
| 10 | 0.32 | 0.65 | 0.65 | 0.33 | 0.65 | 0.28 |
| 12 | 0.43 | 0.50 | 0.64 | 0.33 | 0.67 | 0.33 |
| 14 | 0.45 | 0.60 | 0.65 | 0.33 | 0.31 | 0.13 |
| 16 | 0.19 | 0.26 | 0.30 | 0.12 | 0.29 | 0.13 |
| 18 | 0.19 | 0.28 | 0.29 | 0.12 | 0.31 | 0.12 |
| 20 | 0.02 | 0.29 | 0.30 | 0.12 | 0.16 | 0.12 |

Table 7: The average efficiencies of the parallel matrix multiplication.

Figure 6: A graph of average efficiencies of the parallel version.

|  |  |  |  |
| --- | --- | --- | --- |
| Failure Limit Finding Trials | | | |
| Square Matrix Size | Total Running Time (s) | Total Computation Time (s) | Total Message Passing Time (s) |
| 4800 | 895.340 | 888.261 | 7.079 |
| 5600 | 424.974 | 415.124 | 9.849 |
| 6400 | 1986.352 | 1973.761 | 12.591 |
| 7200 | 902.404 | 886.235 | 16.169 |
| 8000 | 4115.553 | 4095.907 | 19.645 |
| 8800 | 5787.621 | 5763.666 | 23.954 |
| 9600 | 6542.170 | 6513.871 | 28.297 |
| 10400 | 16176.194 | 16142.987 | 33.207 |
| 11200 | 15921.415 | 15882.888 | 38.526 |
| 12000 | 23348.33 | 23304.105 | 44.224 |

Table 8: The total runtimes for the modified SUMMA program with large matrices until its failure using 20 processors. Interestingly, these failure resulted due to TCP requests being unable to be satisfied. It is also interesting how wildly variable the runtimes are – compare 4800 to 5600 and 6400 to 7200.

**Issues**

The speedups were not as high as would be expected for this operation. I am particularly astounded at this considering the fact that there was virtually no message passing time with my algorithm and since the work was divided as evenly as it was. Perhaps if the nodes used all had similar computation power, the grid was not overloaded, etc. this issue would not have presented itself. I suspect that the fact that my algorithm allocated both matrices in their entirety may have suppressed any benefits of working with smaller memory segments as well.