USING OPENMP TO PARELLIZE MATRIX MULTIPLICATION ON A SMP

Jason Glass and Michael Threatt

EECS 453

Table of Contents

Abstract

The purpose of this project was to optimize the Matrix Multiplication (MatMul) algorithm on a shared memory multiprocessor using OpenMP. The Matmul Algorithm has a runtime complexity of O(N^3) and an output complexity of O(N^2). We measured speedup and efficiency on 2,4 and 8 processors compared to an unoptimized uniprocessor baseline. The geometric mean of speedup was 1.89x, 3.657x and 6.95x for 2,4, and 8 cores respectively. The corresponding efficiency is 94.5%, 91.4% and 87.0%.

Background

We use Openmp for experimental purposes. Openmp is an open source library for shared memory multiprocessing. It contains instructions that allow a programmer to control the number of active threads, data chunk size , and scheduling style. Openmp was first released in 1997 for Fortran and was released on C/C++ the following year. A thread is the basic unit to which the operating system allocates processor time. They effectively exist to execute sections of code for our test. Chunk size is the number of iterations given to each thread. If no value is explicitly defined Openmp will use the default action and spread chunks that are roughly the same size across the threads. There are several scheduling styles that Openmp employs: static, dynamic, guided, auto, and runtime.

Static scheduling occurs OpenMP divides the iterations into chunks and it distributes the chunks to threads in a circular order. Dynamic scheduling occurs when threads request chunks and keep requesting until there are none left. This is the more adaptive version of static scheduling. Guided scheduling is very similar to dynamic scheduling but differs on how chunk size is determined. The size of a chunk is the number of unassigned iterations/number of the threads. As a result the size of a chunk decreases over time. Auto scheduling let's the compiler or runtime system determine the scheduling. Runtime scheduling defers scheduling until runtime. If no scheduling type is defined Openmp uses the default setting which is static scheduling. We used static, dynamic, and guided scheduling for testing purposes.

Matrix multiplication is the product of two separate matrices which produces a single matrix as result. It is particularly useful for machine learning applications and linear algebra computations. The dimension of the result matrix is the row of the the first operand matrix and the column of the second operand matrix. For examples a 2x3 matrix multiplied by a 3x1 matrix would produce a 2x1 matrix. The number of rows in the first operand matrix must also be equal to the number of columns in the second operand matrix for matrix multiplication to be possible. Figure 1 below provides a graphic representation of Matmul algorithm we implemented.

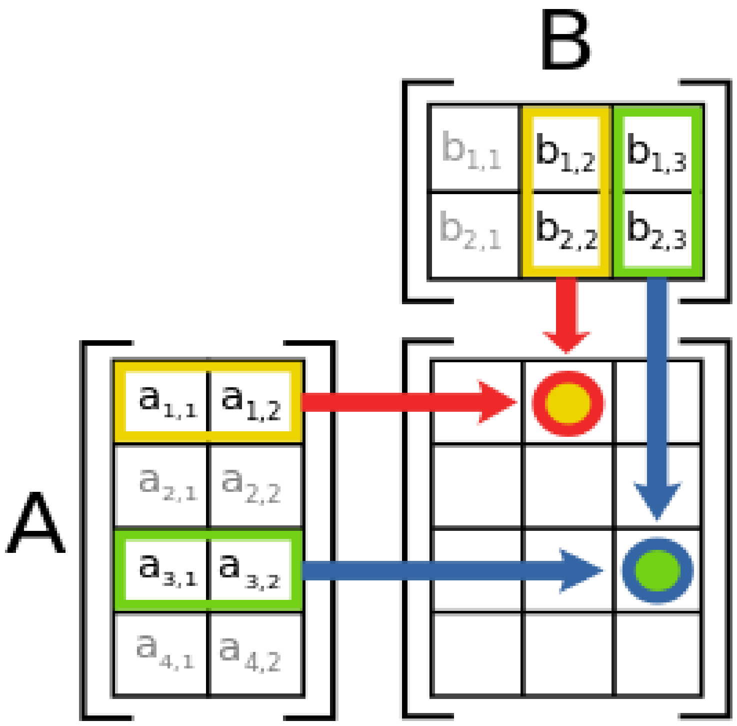
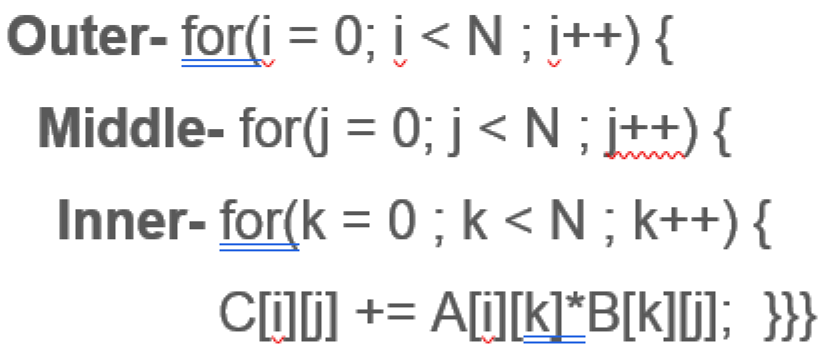


Figure 1: Matrix Multiplication Algorithm

Methodology

For this project we compiled using g++ -o on an AMD Opteron 8218, this architecture is an 8 core SMP with a clock frequency of 2.6 ghz. The L1 is 64K for Data and Instruction caches and the L2 is a shared 1024K cache. We were not able to successfully compile with LLVM or -o2 with g++



```
Outer- for(i = 0; i < N ; i++) {
 Middle- for(j = 0; j < N ; j++) {
  Inner- for(k = 0 ; k < N ; k++) {
          C[i][j] += A[i][k]*B[k][j];  }}}
```

Figure 2: Matrix Multiplication Code

The Matmul Algorithm in Figure 2 consists of 3 nested for loops which each loop running for N iterations, where N is the number of elements per Column and Row, assuming NxN matrix in this paper. Each element of the output Matrix, C, is the dot product of a Row of Matrix A, and a column of Matrix B. The matmul algorithm is an interesting piece of code to parallelize as there is a large variety in choice of granularity. The finest granularity would involve optimizing the calculation of one element of C[i][j], the next step up would be parallelizing the J loop and computing the chunks of one row of C in parallel. The coarsest grain would be computing multiple rows or all in parallel.

Each row of the output matrix can be computed in parallel if you distribute 1 row of Matrix A and All of Matrix B to each processor. This would involve optimizing the algorithm at the Outer loop level, so that each thread has an independent i value, but each will run through j*k inner loop iterations. This implies that the amdahl alpha (serial fraction of the code) is 0% and linear speedup for the code (minus implicit overhead) is possible.

OpenMP pragmas were applied sequentially and various chunk sizes were used for the Guided, Dynamic and Static Scheduling pragmas. The following pragmas were used:

- Shared - Data per execution boundary is shared
- Private - Data per execution boundary is private
- For - parallelize computation across for loop iterations
- Static - Assign thread work in a pre scheduled manner with equal load per thread
- Guided - Threads operate on contiguous chunks of data and chunk size reduces over time
- Chunk - number of iterations assigned per thread, and minimum chunk size for guided scheduling

The following combos of pragma were evaluated:

Table 1: OpenMP Pragmas

Opt#	Optimization
1	#pragma omp parallel for schedule(static) private(i,j,k) shared(X,A,B) num_threads(procs)
2	#pragma omp parallel for schedule(static,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
3	#pragma omp parallel for schedule(static,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
4	#pragma omp parallel for schedule(dynamic) private(i,j,k) shared(X,A,B) num_threads(procs)
5	#pragma omp parallel for schedule(dynamic,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
6	#pragma omp parallel for schedule(dynamic,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
7	#pragma omp parallel for schedule(guided) private(i,j,k) shared(X,A,B) num_threads(procs)
8	#pragma omp parallel for schedule(guided,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
9	#pragma omp parallel for schedule(guided,chunk) private(i,j,k) shared(X,A,B) num_threads(procs)
10	#pragma omp parallel for num_threads(procs) private(i,j,k)
11	#pragma omp parallel for num_threads(procs) private(i,j,k) shared(X,A,B)

The Pragmas in Table 1 were applied at the outermost loop of the Matmul Loop.

Results

Table 2: Runtime, Speedup and Efficiency

Opt#/Cores	1	2	3	4	5	6	7	8	9	10	11
2	10429	9494	9674	9633	9695	9705	9647	9752	9694	10560	10557
4	5177	4937	5194	4904	5230	5250	4808	5226	5112	5218	5219
8	2600	2588	3114	2522	2999	2633	2485	2634	2839	2609	2610
Speedup											
2	1.79	1.97	1.93	1.94	1.93	1.93	1.94	1.92	1.93	1.77	1.77
4	3.61	3.79	3.60	3.81	3.58	3.56	3.89	3.58	3.66	3.58	3.58
8	7.19	7.23	6.01	7.42	6.24	7.10	7.53	7.10	6.59	7.17	7.17
Efficiency											
2	0.90	0.98	0.97	0.97	0.96	0.96	0.97	0.96	0.96	0.89	0.89
4	0.90	0.95	0.90	0.95	0.89	0.89	0.97	0.89	0.91	0.90	0.90
8	0.90	0.90	0.75	0.93	0.78	0.89	0.94	0.89	0.82	0.90	0.90

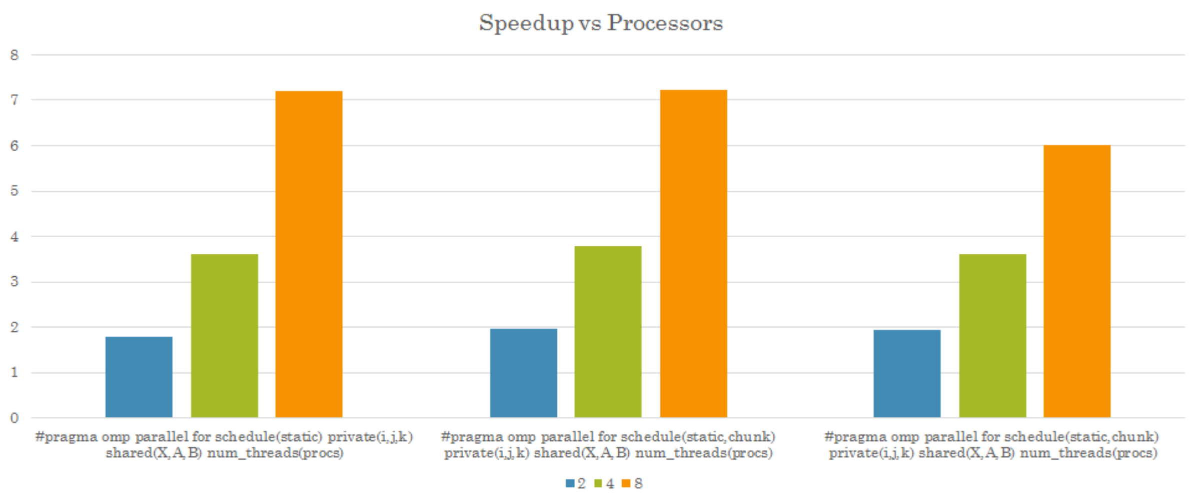


Figure 3: Static Scheduled Speedup

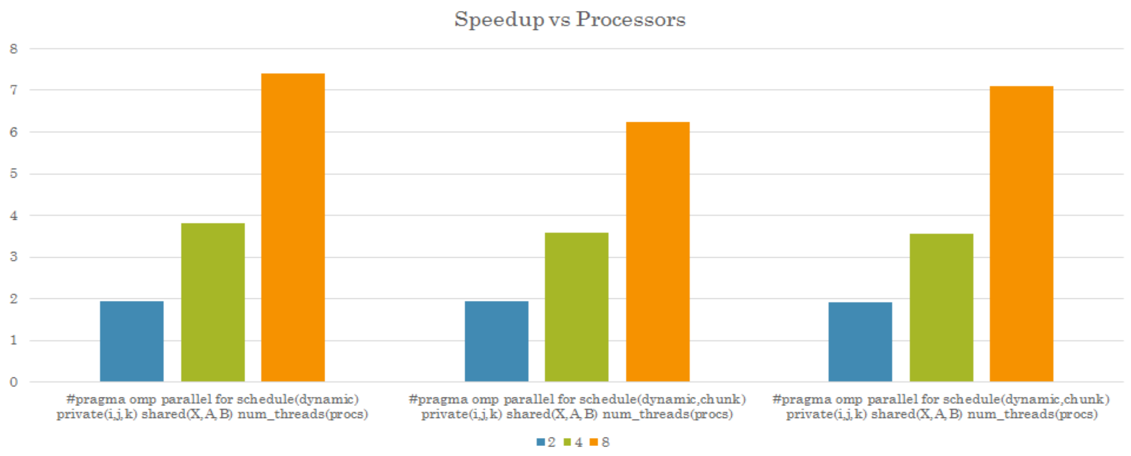


Figure 4: Dynamic Scheduling Speedup

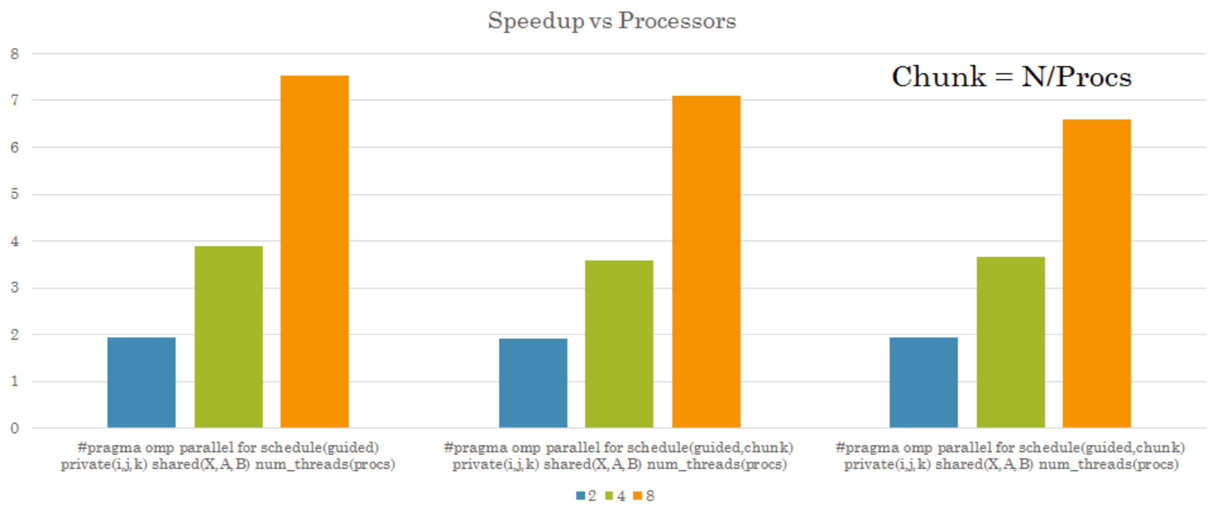


Figure 5: Guided Scheduling Speedup

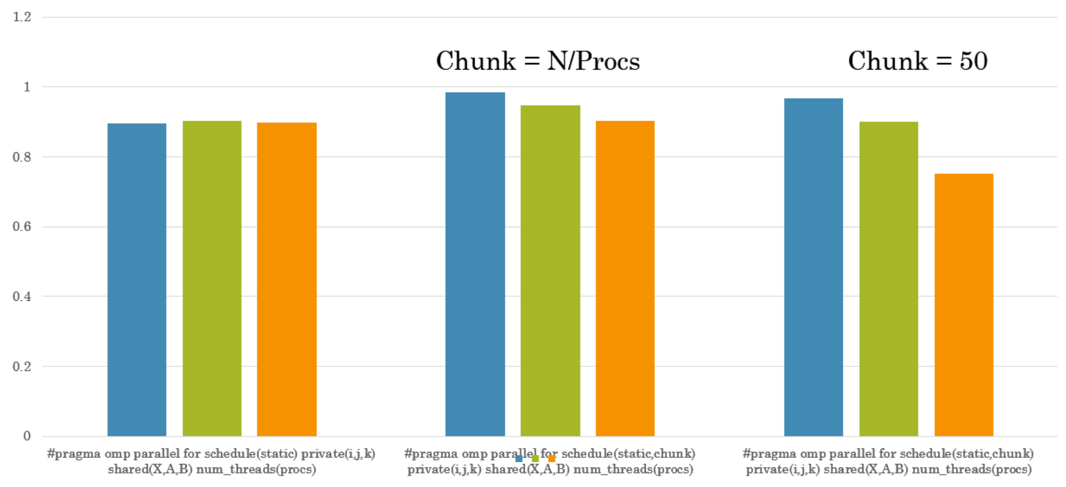


Figure 6: Static Scheduling Efficiency

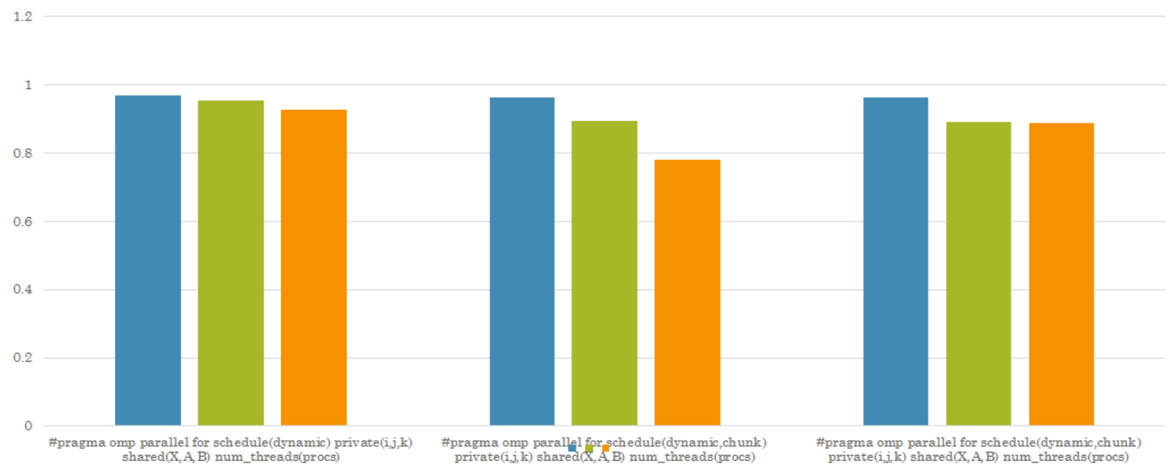


Figure 7: Dynamic Efficiency

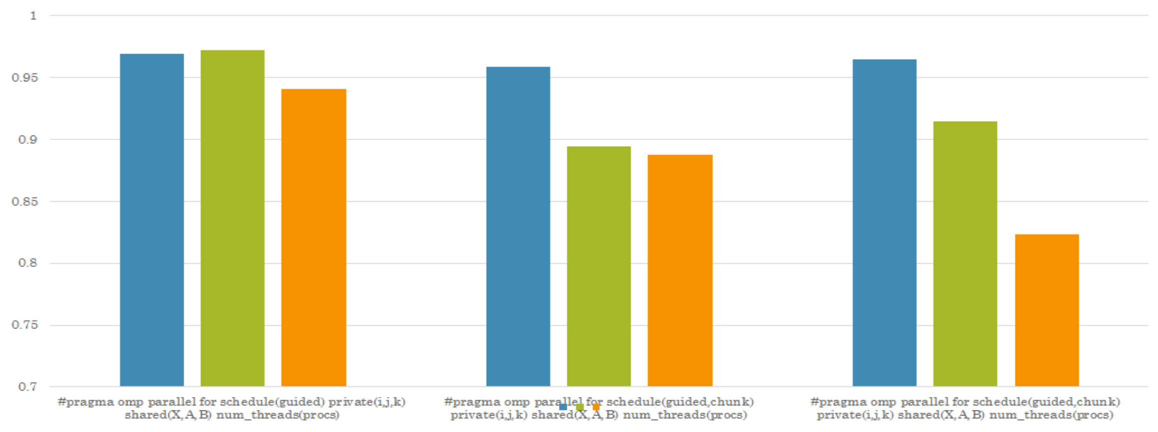


Figure 8: Guided Scheduling Efficiency

Conclusion

This project successfully showed efficient and nearly linear speedup for the Matmul Algorithm using a minimal selection of OpenMP pragmas. The best speedup and efficiency achieved was for the #pragma omp parallel for schedule(guided) private(i,j,k) shared(X,A,B) num_threads(procs), with an average efficiency of 95% . The Matmul algorithm is a good case study for analyzing the speedup from OpenMP due to the algorithm being fully parallelizable, depending on the matrix dimensions being evenly divisible by number of cores, as each row of the output matrix can be computed independently of one another. It can be presumed that most if not all loss in efficiency is due to the OpenMP implicit overhead. This would not be the case for an algorithm like Gaussian Reduction since the amdahl alpha is greater than 0% due to data dependencies.

Future work would include implementing more novel and advanced matmul algorithm optimizations that implement some of the more advanced OpenMP pragma such as collapse and reduction.

Related Work

[1]Alwis, Roshan. “Parallel Matrix Multiplication [C][Parallel Processing].” Medium, Augmenting Humanity, 7 Aug. 2017, medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27.
[2] “Matrix Multiplication Algorithm.” Wikipedia, Wikimedia Foundation, 11 Apr. 2018, en.wikipedia.org/wiki/Matrix_multiplication_algorithm.
[3] Mattson, Tim, and Larry Meadows. “SC08 OpenMP ‘Hands-On’ Tutorial Available.” OpenMP, 7 Nov. 2016, www.openmp.org/uncategorized/sc08-openmp-hands-on-tutorial-available/.
[4] “OpenMP.” Wikipedia, Wikimedia Foundation, 1 June 2018, en.wikipedia.org/wiki/OpenMP.