Math 627 HW 4

Edward LaFemina
University of Maryland, Baltimore County

October 19, 2015

The product of two matrices $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}$ results in a matrix $C \in \mathbb{R}^{m \times n}$ where the ij-th entry is

$$C_{ij} = \sum_{q=0}^{k-1} A_{iq} B_{qj}.$$

In a naive computer implementation of this product, a programmer must use three loops, one for i, j, and q. For matrices with dimensions larger than 1024, the ordering of these loops is significant. The implementation discussed in this paper uses a row-major format for storing matrices in memory where the columns of a matrix are stored consecutively in a contiguous block of memory containing the whole matrix. To obtain best performance, ordering the loops such that j is iterated on the outside loop, i on the inside, and q in the middle proved to be optimal for this method. This ordering resulted in a time of about 361 seconds versus another ordering of $j-i-q$ which resulted in a performance of about 7289 seconds for $m = k = n = 8192$. To determine if there is a less time consuming way to compute this product, three Basic Linear Algebra Subprogram(BLAS) functions were tested.

The first of the BLAS functions we will discuss is the `cblas_ddot` BLAS level 1 function. This function computes the dot product of two vectors returning a scalar. To use this to compute the matrix product, we can think of the product $C = AB$ where C_{ij} is the dot product of the i-th row of A with the j-th column of B. Thus, a program to compute the matrix product in this way requires two loops and again, the order of the loops is significant because of the row-major alignment of our matrices in memory. It was found that having the j loop on the outside and the i loop on the inside improved performance. As shown in Table 1, this BLAS 1 function resulted in slower times than our naive implementation.

The second of the BLAS functions we will discuss is the `cblas_dger` BLAS level 2 function. To use this function to compute the matrix product, we must write the product C as:

$$C = AB = \sum_{q=0}^{k-1} a_q b_q^T.$$

The `cblas_dger` function computes the outer product of the vectors a_q and b_q^T where a_q is the q-th column of A and b_q^T s the q-th row of B. This results in an $m \times n$ matrix. Each iteration of this summation is called a rank-1 update of matrix C which culminates into the product $C = AB$. As shown in Table 2, this method of computing the matrix product is almost an average 13× speedup over the BLAS 1 method and about a 1.2× speedup over the naive implementation.

The third and final BLAS function we will discuss is the `cblas_dgemm` BLAS level 3 function. This function computes the matrix product directly and is specifically optimized to perform the operation in the least amount of time possible. As shown in Table 2, this method of computing the matrix product is by far the best with a speedup of about 5.7× over the naive method, 108× over the BLAS 1 method, and 7× over the BLAS 2 method. For this reason, we recommend the use of the `cblas_dgemm` function when computing matrix products.

To verify that all four of these methods compute the matrix product correctly, we constructed A and B such that we could easily construct the correct result C. For each of the four methods, we stored the result of the product in D. To verify the correctness of the result, we took the Frobenius norm of the difference of C and D. The Frobenius norm $||\cdot||_F$ is computed

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_ij|^2}$$

which is the same as the Euclidean norm of the $m \times n$ vector we get when storing the matrix in a row-major fashion. Thus we can use the Euclidean norm function developed and tested against Matlab results in earlier assignments or use the square root of the value obtained by calling the BLAS level 1 `cblas_dot` function supplying the matrix A for both vectors and treating it as such. We are thus able to determine the correctness of our matrix product functions by computing $||C - D||_F$. A value of 0 indicates that $D = C$ and our method of computing the matrix product is correct. All four methods yielded Frobenius norms of 0.

To test each method of computing the matrix product, we must allocate memory for A, B, C, D and one $m\times n$ vector to hold the difference $C-D$. For testing, we restricted our cases to values of m, k, and n such that $m = k = n$. Thus we have 4 $m \times m$ matrices and 1 $m \times m$ vector of doubles. Since a double requires 8 bytes, our program will use about $5 * 8 * m^2$ bytes of memory. A memory estimation table for different values of $m = k = n$ follows:

m=k=n	size in MB
1024	40
2048	160
4096	640
8192	2560
16384	10240
32768	40960
65536	163840

Table 1. Based off of memory limitations alone, we must restrict our tests to using a maximum $m = k = n$ of 32768 because we can only use up to 63000 MB of memory on the latest model processors available.

The following table(Table 2) reports the time in seconds needed to complete each method of computing the matrix product at different values for $m = k = n$ as well as speedup, however some tests exceeded maximum allowable run times and an estimated values have been put in parenthesis:

Wall Time in minutes						
m	k	n	Naive	BLAS 1	BLAS 2	BLAS 3
1024	1024	1024	<0.01	0.03	<0.01	<0.01
2048	2048	2048	0.08	1.42	0.11	0.02
4096	4096	4096	0.75	16.51	0.87	0.09
8192	8192	8192	6.02	139.42	8.00	0.76
16384	16384	16384	(41)	(788)	76.71	7.31
32768	32768	32768	(342)	(6498)	(421)	60.17

Speedup						
m	k	n	Naive	BLAS 1	BLAS 2	BLAS 3
1024	1024	1024	1	0.203	0.796	3.711
2048	2048	2048	1	0.057	0.721	3.600
4096	4096	4096	1	0.045	0.859	7.967
8192	8192	8192	1	0.043	0.753	7.876
16384	16384	16384	1	(0.047)	(0.761)	(7.987)
32768	32768	32768	1	(0.047)	(0.761)	(7.999)

To achieve even greater speedup, we wish to compute the matrix product in parallel. To do this, we recognize that our third BLAS function computes successive outer products of subsections of the matrix and uses the sum of the outer products to produce the final result. As defined in the Intel Math Kernel documentation, the `cblas_dgemm` function performs the following operation:

$$C := \alpha * AB + \beta * C.$$

We notice that when $\alpha = 1$ and $\beta = 0$ we get our desired matrix product AB and internally, the function computes smaller outer products of pieces of A and B that can fit in the CPU's cache. Because the order of addition to C does not matter, we can split A and B ourselves across several processes, perform the `cblas_dgemm` function with l_A, l_B, and $local\_C$. As long as $l\_A \in \mathbb{R}^{m \times l\_k}, l\_B \in \mathbb{R}^{l\_k \times n}$ for some $l\_k = \frac{k}{np}$ we will get a resultant $local\_C \in \mathbb{R}^{m \times n}$ whose ij-th entry is

$$local\_C_{ij} = \mathbf{l\_a}_i \mathbf{l\_b}_j^T = \sum_{q=0}^{l\_k-1} a_{iq} b_{qj}, \;\; 0 \le i,j < l\_k, \mathbf{l\_a}_i \in l\_A, \mathbf{l\_b}_j \in l\_B.$$

We notice that

$$C = AB = \sum_{q=0}^{k-1} \mathbf{a}_i \mathbf{b}_j^T = \sum local\_C, \;\; 0 \le i,j < k, \mathbf{a}_i \in A, \mathbf{b}_j \in B.$$

Thus we can compute *local_C* on different processes and the sum of all *local_C* will be the desired matrix product C. To verify this computation, we again checked that the Frobenius norm of the difference of the computed value and the correct result which we know by construction of A and B. In all cases, the Frobenius norm was 0 and we are confident we have a correct implementation of a matrix product.

To distribute the rows and columns of A and B across np processes, we created MPI derived datatypes. We created a datatype for columns using `MPI_Type_vector` because our matrix B is stored in a row-major format and thus each element of a row B is exactly k away from the previous element. We used `MPI_Type_contiguous` to create a datatype for the columns of A because they are stored next to each other. We then used `MPI_Send` and `MPI_Recv` to distribute *l_k* rows and columns across np processes and performed our calculations of *local_C*. We then used `MPI_Reduce` to sum all the *local_C* values on process 0 and report our results.
Table 3. Time in seconds taken to complete the matrix product. Everything run with 2 processes per node except $p = 1$. Because process 0 needs to hold matrices A, B, C, and D and all other processes hold less than process 0, our memory estimates in Table 1 hold for these tests as well.

Wall Time in seconds							
m	k	n	p=1	p=2	p=4	p=8	p=16
1024	1024	1024	0.1099	0.0627	0.053	0.0439	0.0386
2048	2048	2048	1.4378	1.4226	1.1698	0.2147	0.1671
4096	4096	4096	6.4344	4.1432	2.2738	1.346	0.7791
8192	8192	8192	46.0086	24.5055	14.1305	8.2525	5.1457
16384	16384	16384	440.8183	222.2572	118.3714	63.941	37.1295
32768	32768	32768	-	-	-	-	-

Speedup							
m	k	n	p=1	p=2	p=4	p=8	p=16
1024	1024	1024	1	1.7527	2.0735	2.5034	2.8471
2048	2048	2048	1	1.0106	1.2290	6.6967	8.6044
4096	4096	4096	1	1.5530	2.8298	4.7803	8.2587
8192	8192	8192	1	1.8774	3.2559	5.5751	8.9411
16384	16384	16384	1	1.9833	3.7240	6.8941	3.1880
32768	32768	32768	1	-	-	-	-