CS 140 Homework 4:
SUMMA Matrix Multiplication

Assigned February 7, 2007

Due by 11:59 pm Thursday, February 21

This assignment is to write a high-performance routine for parallel multiplication of one dense matrix by another. The methods we've seen in class all have some disadvantages; the block column algorithms do more communication than necessary, and Cannon's systolic algorithm is appealing but uses too much memory and is hard to generalize to the case where the matrix dimensions aren't divisible by the square root of the number of processors. In this assignment you will implement and experiment with the so-called "SUMMA" algorithm, which is the one that is used in practice in Scalapack and related parallel libraries.

1 The SUMMA Algorithm

The basic algorithm is pretty simple. It's described in pictures in the online course slides from the January 31 lecture (slides 31-35). The original paper about it is also online (see the link under Homework Assignments: Homework 4 on the course web page). That paper is a little hard to read, partly because it describes a more general and complicated primitive.

Here it is in words: Suppose we want to multiply two n-by-n matrices A and B, resulting in matrix C. Using the fact that

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j),$$

we perform a sequential loop for k from 1 to n, at each step of which we update all the $C(i,j)$ values. In Matlab notation, we implement the following:

```
C = zeros(n,n);
for k = 1:n
   C = C + A(:,k) * B(k,:);
end
```

Here `A(:,k)` is column k of A, and `B(k,:)` is row k of B.

It remains to describe how to do a single iteration of the loop, i.e. a single update to C, in parallel. The first question (as always) is, where's the data? We will think of the processors as forming a two-dimensional grid (with p_r rows and p_c columns in the picture on the slides). Each of the matrices A, B, and C is divided into blocks, one block per processor, each block having size approximately n/p_r by n/p_c.

At iteration number k, each processor needs to update its own block of C, for which it requires just part of column `A(:,k)` and part of row `B(k,:)`. Therefore, at the beginning of iteration k, each of the p_r processors that owns part of column `A(:,k)` sends that partial column to each of the other processors in its row of the grid, and similarly each of the p_c processors that owns part of row `B(k,:)` sends that partial row to each of the other processors in its column of the grid.

One nice thing about SUMMA is that it doesn't require that p_r and p_c be the same, or that n be divisible by either one, or even that the matrices A, B, and C be square. However, for this homework assignment, it's okay to assume that $p_r = p_c = \sqrt{p}$, the matrices are all n-by-n, and n is divisible by $\sqrt{p}$.

2 Improvements

There are two ways to improve on the basic algorithm.

The first is to work with blocks of rows and columns instead of individual rows and columns. Instead of doing n iterations of the outer loop, each of which uses one column of A and one row of B to do a rank-1 update to C, we pick some blocksize b (32, for example), and we do n/b iterations of the outer loop, each of which uses b columns of A and b rows of B to do a rank-b update to C. The Matlab pseudocode then becomes

```
C = zeros(n,n);
for k = 1:b:n                        % k goes from 1 to n in steps of b
   kk = k:(k+b-1);                   % kk is [k k+1 k+2 ... k+b-1]
   C = C + A(:,kk) * B(kk,:);
end
```

Now the inner loop involves multiplying the n-by-b matrix `A(:,kk)` by the b-by-n matrix `B(kk,:)`, which gives an n-by-n matrix that is added to C.

This improves things in two ways. First, it requires fewer communication actions. The same total volume of communication is done, but it's done with a factor of b fewer messages. Second, the computation on each processor now involves multiplying two matrices (an n/p-by-b piece of A and a b-by-n/p piece of B) instead of two vectors. This sequential matrix multiplication can use blocking (as described in class) to improve its cache behavior. Rather than trying to write a well-blocked sequential matrix multiplication code yourself, I recommend that you just call the sequential BLAS library code "`dgemm`" to do this multiplication. (`dgemm` is part of the ESSL library on DataStar.)

The second improvment to the basic SUMMA algorithm is in the communication pattern. At a particular step k, how does a processor that has a piece of `A(:,k)` send it to the other $\sqrt{p}-1$ processors that need that piece?

One way is to do $\sqrt{p}-1$ calls to `MPI_Send` with appropriate destinations. A second way is to do a tree of sends—one processor sends to a second, then each of them sends to another, then each of those four sends to another, until all $\sqrt{p}$ have gotten the message. This is the same volume of communication, but there's not the bottleneck of one processor doing $\sqrt{p}$ sends; nobody does more than $\log\sqrt{p} = (\log p)/2$ sends.

A third way is to use the `MPI_Bcast` primitive, but instead of using the `MPI_COMM_WORLD` communicator that includes all the processes, set up communicators that include just the processes in one row or one column of the processor grid. This is a bit complicated, but it might give the best performance. If you'd like to try it, read Section 5 of the Pacheco MPI tutorial that's linked to the course web page, looking at the description of `MPI_Cart_sub` and its relatives. (Note that Fox's algorithm for matrix multiplication, which Pacheco uses as an example, is *not* the same as SUMMA.)

3 What Experiments to Do

Implement the SUMMA algorithm as described above. For full credit, you should implement the blocked version, using a sequential BLAS library version of `dgemm` for local matrix multiplications.

Experiment with matrix dimensions n ranging from very small up to as large as practical. You should first get your program running correctly on one processor! Then, experiment with $p = 1, 4, 9, 16$ and possibly higher. Experiment with various block sizes b from 1 up to at least 32. It's okay to use matrix sizes n that are multiples of $\sqrt{p}$, and to use block sizes b that are even divisors of $n/\sqrt{p}$.

For test matrices, I suggest the following three types.

1. Bi/tridiagonal matrices. Let A be a matrix that has all elements on its main diagonal equal to 1, all elements on its subdiagonal equal to 1, and all other elements equal to 0. Let B be the transpose of A. Then the product $A \times B$ has the following pattern, which you can check to verify that your code is correct.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

2. Hadamard matrices. A Hadamard matrix is a matrix whose entries are all ± 1 and whose rows are mutually orthogonal. If H is a symmetric n-by-n Hadamard matrix, then $H \times H$ is a matrix whose diagonal elements are all equal to n and whose off-diagonal elements are all equal to 0. Symmetric Hadamard matrices make good test cases for your code, since it's easy to test whether you are getting the right answer.

   See the Wikipedia entry for "Hadamard matrix" for a simple recursive way to construct a symmetric n-by-n Hadamard matrix when n is a power of 2. We will put some small Hadamard matrices on the course web site.

3. Random matrices. Use one of the random number generators from Homework 1 to generate random values for the matrices A and B. In this case you can generate the matrices in parallel; you don't have to send them out from the root process. There's no easy way to check the correctness of your program for this type—you should do correctness checks on the other two test matrix types—but you can use it for timing benchmarks.

You should organize your code so that the main program first calls one routine called "`setup`" that sets up the matrices A and B (with their data distributed among the parallel processes); and a second routine called "`matmul`" that does the multiplication, resulting in a distributed matrix C; and a third routine called "`evaluate`" that checks to make sure that the answer is right (for matrix types (1) and (2) above). For your timing results, you should only measure the time of the `matmul` routine.

There will be some extra credit for experimenting with different communication structures, both for the tree of sends and for MPI broadcasts along rows and columns of the processor grid. Any comparisons you can make about the efficiency of the various communication structures will be very interesting.

There will also be a little extra credit for finding the PBLAS parallel library routine `pdgemm` and comparing its running time to your code. `pdgemm` is part of the PESSL library on DataStar. It might only work across processors in a single node; I'm not sure.

There will also be some extra credit for the fastest and the second-fastest results on the "CS 140 Benchmark" on DataStar.

Here are the rules for the CS 140 Benchmark: You must use matrix type (3), random matrices. You must use exactly 16 processors in a 4-by-4 grid. You choose n, the matrix dimension. It must be at least 4000, and may be as large as you want. (However, the total run time of your code should be at most 2 minutes.) The matrix data must be divided evenly among the processors. Your score is the measured wall clock time in seconds of your `matmul` routine on n-by-n matrices divided into $2n^3$. That is, your score is your code's measured flops-per-second. You should include your own flops-per-second results in your report, but the prize will be based on our own runs with your code. If you want to be considered for this prize, you should say so in your README and also include everything we need to compile your code (Makefiles, command lines, etc.) just the way you did.

4 What to Turn In

Turn in

- Your source code.
- A `Makefile` that compiles your code.
- A `README` file that contains instructions for compiling and running your code, plus tables of your run time results, plus a report that describes and interprets your results and any conclusions you draw from them.
- If you want us to test your code for the CS 140 Benchmark prize, please say so clearly in your `README` file and include clear instructions for how to run the code and for what values of n to use.
- If you worked in a team, your `README` should include the names and perm numbers of both members.

You may do this assignment alone or in groups of two.