SUMMA Matrix Multiplication

Due 12:29 pm Oct 30

In this assignment you will implement the SUMMA algorithm for multiplying two dense matrices. SUMMA algorithm is used in practice in Scalapack and related parallel libraries.

1 The SUMMA Algorithm

The lecture notes include detailed information about the algorithm and the pseudo code.

You are encouraged to read the original SUMMA paper.
(http://www.netlib.org/lapack/lawnspdf/lawn96.pdf)

The basic SUMMA algorithm is pretty simple. Here is the algorithm in words: suppose we want to multiply two n-by-n matrices A and B, resulting in matrix C. The standard sequential algorithm (without cache blocking) is usually written as three nested loops on i (row number in C and A), then j (column number in C and B), and then k (inner index, column of A and row of B) in order. However, we can nest the loops in any order and still get the same answer. Using the order kij instead of ijk, we get the following code:

```
for ( k = 0; k < n; k++)
       for ( i = 0; i < n; i++)
              for (j = 0; j < n; j++)
                     C[i][j] = C[i][j] + A[i, k] * B[k, j]
                       //innermost loop: A, column; B, row
```

For each iteration of k, the product of a column vector A times a row vector B is an n-by-n matrix, actually just the "multiplication table" of the elements of the two vectors. Placing k as the outmost loop is the same as expressing C as the sum of n of those "multiplication table" matrices. The SUMMA algorithm runs the k loop sequentially, but each iteration of k is parallelized on a two-dimensional grid of processors.

How is a single iteration of the k loop, i.e. a multiplication table updated to C, performed in parallel?

First, we need to decide where the data is and how the computation is decomposed. We will think of the np processors that form a two-dimensional grid (with q rows and q columns where $np=q^2$ in the picture on the slides). Each of the full matrices A, B, and C is divided into blocks, each block with size approximately n/q by n/q. The algorithm distributes one block (n/q by n/q) of each of the matrices A, B and C to one processor, and decompose the computation in such a way that each processor calculates the corresponding block of C.

Second, at iteration number k, each processor needs to update its own block of C, for which it requires a block of A and a block of B. Therefore, at the beginning of iteration k, each of the q processors that owns the block of A sends the block to each of the other processors in its row of the grid, and similarly each of the q processors that owns the block of B sends the block to each of the other processors in its column of the grid. Once each processor has the blocks of A and B, it can update C once.

All blocks of C will be updated q times and then complete at the end.

One nice thing about SUMMA is that it doesn't require that the number of rows and the number of columns in the processor grid be the same, or that n be divisible by either one, or even that the matrices A, B, and C

be square. For simplicity, for this homework assignment, we can assume that $q = sqrt(p)$, the matrices are all n-by-n, and n is divisible by q.

2 Implementation

An incomplete program summa.c can be downloaded from canvas. It has the main function and helping functions to allocate spaces for matrices and initialize them. You can start from this program.

For the implementation, you will replace "…" with code segments to set up the MPI processes, the processor grid, matrix blocks, and timing in the main function, and implement the SUMMA algorithm in the matmul function. The resulting matmul function should have the exact structure as the pseudo code in the lecture notes.

The SUMMA algorithm broadcasts blocks of A to processes in the same rows and blocks of B to processes in the same columns. You will need to set up communicators that include just the processes in one row or one column of the processor grid using MPI_Cart_sub and its relatives.

I have posted on canvas two example programs that can be helpful to this assignement: mpi_matmul.c and cartesian.c. The former shows how to send and receive matrices. Pay special attention to how the matrix is specified in the MPI_Send and MPI_Recv functions. The latter shows how to create row and column communicators.

3 Experiments

Complete the summa.c program as indicated above.

I suggest to use the bi/tridiagonal matrices to test if the program is running correctly. 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 × B has the following pattern, which you can check to verify that your code is correct. A is:

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

B is:

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

C is

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

You should use random matrices when you collect experimental data for the report. Use one of the random number generators to generate random values for the matrices A and B. As 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 with the bi/tridiagonal matrices first.

Collect the execution time of your program with various processor counts and matrix sizes. Use square number such as 1, 4, 9, 16, 25, 36, 49, 64 for the number of processors. The matrix size may be as large as you want but at least 4000. A single run with the largest processor count should be in the order of several minutes.

4 Submission

Your writeup includes the experimental platform, instructions for compiling and running your code, collected experimental results with tables and graphs, observations and discussions on the results, and any conclusions you draw.

Submission will be through canvas. Please place the following files in your submission:

1. Your writeup, in a file called writeup.pdf
2. Your source codes in a single tar file. All code must be compilable and runnable!