Distributed Memory Matrix Multiplication

Design and Analysis of Algorithms
for Parallel Computer Systems
Assignment 2

1. You should complete this assignment in **groups of at most 2 students**.
2. Write the **name** and **email** of every group member on the front page.
3. Include the **path to the source** in your report and append a printout.

Good Luck!

1 Introduction

Matrix multiplication is a common operation in most dense linear algebra algorithms and there are efficient implementations available online. Because of the regularity of the algorithm and the many ways it may be organized it is a perfect example in education. The aim of this assignment is to get you familiar with 2D block cyclic data layout and investigate the potential benefits of overlapping communication with computation. You will implement a variant of an algorithm for distributed memory matrix multiplication known as SUMMA. You will also evaluate how overlap of communication and computation might or might not increase the performance of the algorithm.

The matrix multiply operation occurs in many algorithms in different forms. The Basic Linear Algebra Subprograms (BLAS) has been an ongoing effort to standardize a set of interfaces to common linear algebra operations. The double precision matrix multiply is a Level-3 BLAS operation and the routine is called `DGEMM` (Double precision GEneral Matrix Multiply and add) and performs the generic operation

$$C \leftarrow \alpha \, \mathrm{op}(A) \, \mathrm{op}(B) + \beta C$$

where $\mathrm{op}(X) = X$ or $\mathrm{op}(X) = X^T$ and α, β are double precision scalars. In other words, this means that A and/or B may be implicitly transposed.

2 Data Layout and Distribution

The matrices A, B, and C have dimensions $m \times k$, $k \times n$, and $m \times n$, respectively. The dimension m is blocked with blocking factor mb, n with nb, and k with kb. The matrices are distributed using a 2D block cyclic distribution and their first elements all map to process $(0, 0)$. See Figure 1 for an example where the

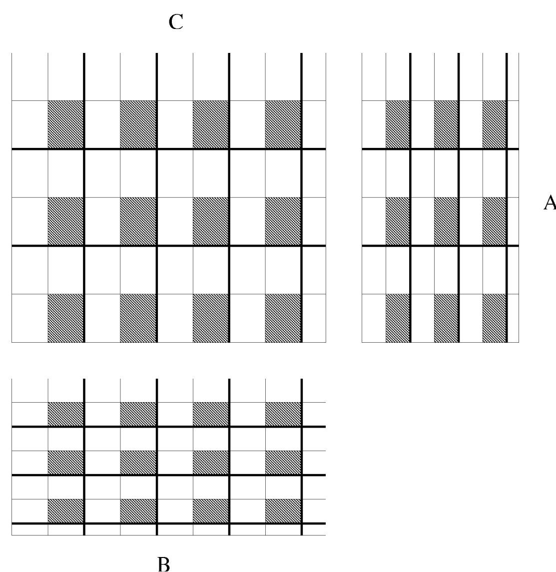


Figure 1: Distribution of the matrices on a 2-by-2 process mesh.

blocks stored at process $(1, 1)$ are highlighted. Due to the alignment restriction

the highlighted blocks in A and B are aligned with those in C which simplifies communication.

3 Algorithm

The algorithm is sometimes referred to as the SUMMA algorithm but was independently implemented by at least three groups, among them Fred Gustavson. It is particularly useful when the matrices have a 2D block cyclic data layout. Implementations of the algorithm require little extra memory and communication may be overlapped with computation.

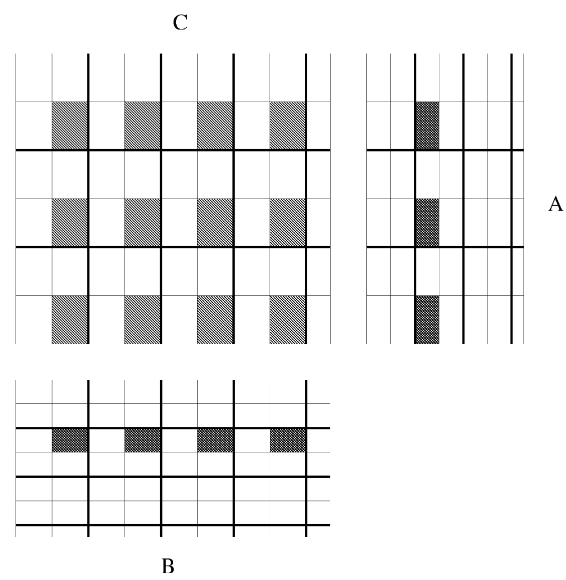
The *basic algorithm* (without overlap) is as follows.

```
for j = 1 to k step kb
   blksz = min(kb, k - j + 1)
   Broadcast A(:, j:j+blksz-1) on rows into E buffer
   Broadcast B(j:j+blksz-1, :) on columns into S buffer
   Update C = C + E*S
end for
```

The algorithm steps through the block columns (rows) of A (B) and in each step a blocked outer-product update on C is performed.

In Figure 2, we illustrate a snapshot of the algorithm as observed by the $(1, 1)$ process. All of the local part of C is updated (light gray) by using parts of the block column of A and block row of B (dark gray). The local view is for the $(1, 1)$ process only (dimensions of local matrices vary across the mesh). The relations between the dimensions of A and the dimensions of the S and E buffers is hopefully apparent from the figure. The buffers are used to store the matrices used for update since it is often remote and is brought in by explicit messages.

Overlap of communication with computation can be achieved by keeping two sets of S and E buffers and alternating between them after each iteration. Communication for the upcoming iteration is performed while the current iteration is in the update phase.



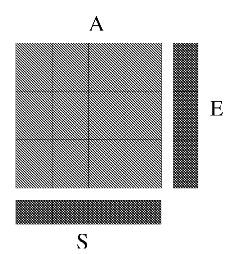


Figure 2: Snapshot of the algorithm at `j = 1 + 2*kb`.

4 Tasks

Your ultimate task is to implement, evaluate, and compare two algorithms: the basic algorithm and the overlapped algorithm. To help you get started we provide a sequence of smaller tasks that you may use as a guide towards completing this assignment.

1. Get some practice with the 2D block cyclic distribution. Sketch some small examples on paper and learn how to map global to local indices and how to determine which process owns a particular global index. Also learn how to compute the size of the local matrices.

2. Choose a programming language (Fortran or C) and familiarize yourself with writing, compiling, and executing MPI applications on the Sarek cluster. C has the advantage of being a familiar language to many of you, whereas Fortran has a powerful syntax for matrices and arrays very similar to MATLAB.

3. Code a test framework that allocates/deallocates the local matrices for A, B, and C and fills them with some initial values. Also provide code for measuring the wall-clock time of your routines.

4. Read the documentation of the `DGEMM` Level-3 BLAS routine for doing local optimized matrix multiply. Make sure you know how to interface to the `DGEMM` routine from the programming language you have chosen.

5. Implement the *basic algorithm* using `MPI_Bcast` for the broadcasts. Observe that you need to split (`MPI_Comm_split`) the communicator along the rows and columns. Insert code to measure the combined time spent in communication and computation during the execution of the algorithm.

6. Implement the *overlapped algorithm* by using two alternating S and E buffers. Communication must now be performed using asynchronous point-to-point messages (`MPI_Isend`, `MPI_Irecv`) since asynchronous collectives are not part of MPI. Also here you need to insert code to measure the communication and computation costs.

7. Test both algorithms for various block sizes, dimensions, and process meshes. Recommended block sizes are in the range of 50 to 100, dimensions $m = n = k = 1000$ to 4000, and process meshes 2×2 up to at least 4×4. Record the parallel execution time, the communication cost, and the computation cost for each test.

8. Do performance analysis on your data and pay particular attention to *strong scalability*, i.e., how the performance per process changes when the problem size is fixed while the number of processes increases. Is one algorithm always better than the other or can you find two sets of parameters for which the best algorithm differs?

5 Random Tips and Tricks

1. Matrices stored as 2D arrays are referenced in Fortran like `A(i,j)` to get element a_{ij} and in C like `A[i-1][j-1]` to get the same element. Since the memory is one dimensional the compiler must map the 2D matrices into 1D memory and this can be done in many different ways. In Fortran the elements are stored column-by-column (the first index varies the fastest) whereas in C the elements are stored row-by-row (the last index varies the fastest). When referencing a submatrix in an array stored in either fashion it is necessary to specify the *leading dimension* which is the increment between elements in adjacent columns (Fortran) or rows (C). These leading dimensions must be specified when calling BLAS routines and they are always named `LD_` where `_` is the name of the matrix (e.g., LDA for the leading dimension of the matrix A).

2. If you intend to use C we recommend that you use 1D arrays and do the 2D addressing yourself to get column-by-column storage.

3. On Sarek you may load these modules:

   ```
   module add mpich/pgi    # (Compiler and MPI implementation)
   module add libgoto      # (Optimized BLAS implementation)
   ```

   and use the `mpif90` (Fortran) or `mpicc` (C) compiler. To link with GotoBLAS you need to add `-lgoto`.

4. One way to get accurate wall-clock times is to use the following snippet.

   ```
   #include <sys/time.h>
   #include <stdlib.h>
   double wallclock() {
      struct timeval tv;
      gettimeofday(&tv, NULL);
      return (double)tv.tv_sec + 1.E-6*tv.tv_usec;
   }
   ```

5. Documentation for `DGEMM` can be found in the reference BLAS implementation available at `http://www.netlib.org/blas/dgemm.f`.

6. If you call `DGEMM` from C you should be aware that interfacing Fortran from C is highly compiler specific. If you use the recommended PGI compiler then this can be assumed:

   - the symbol is named `dgemm_`,
   - all arguments must be pointers (`char*`, `double*`, `int*`) which in particular means that you can not pass the scalars α, β as literal constants.

7. The MPI v1.1 standard document in HTML format: `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

8. ScaLAPACK has several auxiliary routines that are useful when working with 2D block cyclic data layouts. Take a look at the `numroc` function and the `infog2l` subroutine at `http://www.netlib.org/scalapack/tools/`.