Notes on Parallelizing Matrix-Matrix Multiplication

Robert van de Geijn, Martin Schatz, Tze Meng Low

April 16, 2013

1 A simple implementation

In this assigment, you start with a very simple parallel matrix-matrix multiplication that implements $C = AB + C$ as a sequence of rank-k updates:

$$C = \left(\begin{array}{c|c|c|c} a_0 & a_1 & \ldots & a_{k-1} \end{array} \right) \left(\begin{array}{c} \hat{b}_0^T \\ \hat{b}_1^T \\ \vdots \\ \hat{b}_{k-1}^T \end{array} \right) = ((\cdots((C + a_0\hat{b}_0^T) + a_1\hat{b}_1^T) + \cdots) + a_{k-1}\hat{b}_{k-1}^T)$$

1.1 Distribution

Consider Figure 1–3, which illustrate how matrices C, A, and B are distributed to a $r \times c$ mesh of (MPI) processes, when $r = c = 3$. They are distributed using an elemental cyclic distribution.

1.2 Routine `ParallelRank1`

Since the matrix multiplication can be accomplished as a loop around rank-1 updates, I created a routine `ParallelRank1` that implements a rank-1 update with the pth column A multiplying the pth row of B, updating C.

Notice that this routine communicates the pth column of A within rows of nodes and the pth row of B within columns of nodes to achieve the duplication of data illustrated in Figure 2. The duplicated data are in the work arrays `work_A` and `work_B`, respectively, and the local computation is then performed by the call to the BLAS routine `dger_`, which performs a rank-1 update. Notice that `dger_` uses a Fortran interface, meaning that data must be passed by address, which explains some of the ugliness...

1.3 Routine `ParallelMMult`

The routine `ParallelMMult` in file `ParallelMMult_1.c` simply implements the parallel matrix-matrix multiplication as a loop around calls to `ParallelRank1`.

1.4 The driver routine

The driver routine sets up the MPI environment, creates random matrices, copies (via the utility routine `CopyMatrixGlobalToLocal`) the local part of those random matrices into the local arrays used for the parallel matrix-matrix multiplication, etc. It uses utility routines to create a random matrix and to compare the contents of two matrices.

1.5 Executing

Copy `~rvdg/class/CS378S13/PMMult` into a directory of your own. Executing `make run` will compile and test (with a 2×3 mesh of processes). The reported `diff` should be small.

$$
\begin{array}{|c|c|c|}
\hline
\begin{array}{ccccc} & \beta_{4,0} & \beta_{4,3} & \beta_{4,6} & \cdots \\ \alpha_{0,4} & \gamma_{0,0} & \gamma_{0,3} & \gamma_{0,6} & \cdots \\ \alpha_{3,4} & \gamma_{3,0} & \gamma_{3,3} & \gamma_{3,6} & \cdots \\ \alpha_{6,4} & \gamma_{6,0} & \gamma_{6,3} & \gamma_{6,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,1} & \beta_{4,4} & \beta_{4,7} & \cdots \\ \alpha_{0,4} & \gamma_{0,1} & \gamma_{0,4} & \gamma_{0,7} & \cdots \\ \alpha_{3,4} & \gamma_{3,1} & \gamma_{3,4} & \gamma_{3,7} & \cdots \\ \alpha_{6,4} & \gamma_{6,1} & \gamma_{6,4} & \gamma_{6,7} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,2} & \beta_{4,5} & \beta_{4,8} & \cdots \\ \alpha_{0,4} & \gamma_{0,2} & \gamma_{0,5} & \gamma_{0,8} & \cdots \\ \alpha_{3,4} & \gamma_{3,2} & \gamma_{3,5} & \gamma_{3,8} & \cdots \\ \alpha_{6,4} & \gamma_{6,2} & \gamma_{6,5} & \gamma_{6,8} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \\
\hline
\begin{array}{ccccc} & \beta_{4,0} & \beta_{4,3} & \beta_{4,6} & \cdots \\ \alpha_{1,4} & \gamma_{1,0} & \gamma_{1,3} & \gamma_{1,6} & \cdots \\ \alpha_{4,4} & \gamma_{4,0} & \gamma_{4,3} & \gamma_{4,6} & \cdots \\ \alpha_{7,4} & \gamma_{7,0} & \gamma_{7,3} & \gamma_{7,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,1} & \beta_{4,4} & \beta_{4,7} & \cdots \\ \alpha_{1,4} & \gamma_{1,1} & \gamma_{1,4} & \gamma_{1,7} & \cdots \\ \alpha_{4,4} & \gamma_{4,1} & \gamma_{4,4} & \gamma_{4,7} & \cdots \\ \alpha_{7,4} & \gamma_{7,1} & \gamma_{7,4} & \gamma_{7,7} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,2} & \beta_{4,5} & \beta_{4,8} & \cdots \\ \alpha_{1,4} & \gamma_{1,2} & \gamma_{1,5} & \gamma_{1,8} & \cdots \\ \alpha_{4,4} & \gamma_{4,2} & \gamma_{4,5} & \gamma_{4,8} & \cdots \\ \alpha_{7,4} & \gamma_{7,2} & \gamma_{7,5} & \gamma_{7,8} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \\
\hline
\begin{array}{ccccc} & \beta_{4,0} & \beta_{4,3} & \beta_{4,6} & \cdots \\ \alpha_{2,4} & \gamma_{2,0} & \gamma_{2,3} & \gamma_{2,6} & \cdots \\ \alpha_{5,4} & \gamma_{5,0} & \gamma_{5,3} & \gamma_{5,6} & \cdots \\ \alpha_{8,4} & \gamma_{8,0} & \gamma_{8,3} & \gamma_{8,6} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,1} & \beta_{4,4} & \beta_{4,7} & \cdots \\ \alpha_{2,4} & \gamma_{2,1} & \gamma_{2,4} & \gamma_{2,7} & \cdots \\ \alpha_{5,4} & \gamma_{5,1} & \gamma_{5,4} & \gamma_{5,7} & \cdots \\ \alpha_{8,4} & \gamma_{8,1} & \gamma_{8,4} & \gamma_{8,7} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} &
\begin{array}{ccccc} & \beta_{4,2} & \beta_{4,5} & \beta_{4,8} & \cdots \\ \alpha_{2,4} & \gamma_{2,2} & \gamma_{2,5} & \gamma_{2,8} & \cdots \\ \alpha_{5,4} & \gamma_{5,2} & \gamma_{5,5} & \gamma_{5,8} & \cdots \\ \alpha_{8,4} & \gamma_{8,2} & \gamma_{8,5} & \gamma_{8,8} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \\
\hline
\end{array}
$$

Figure 1: Distribution of C to a 3×3 mesh of processes. Also shown is how a_p and $\hat{b}_p$ are duplicated in order to perform the update $C = C + a_p \hat{b}_p$ as local rank-1 updates.

1.6 Comment

When we run this on a real machine (next week), you will see that this implementation is (weakly) scalable, but achieves very low performance.

2 Optimization 1

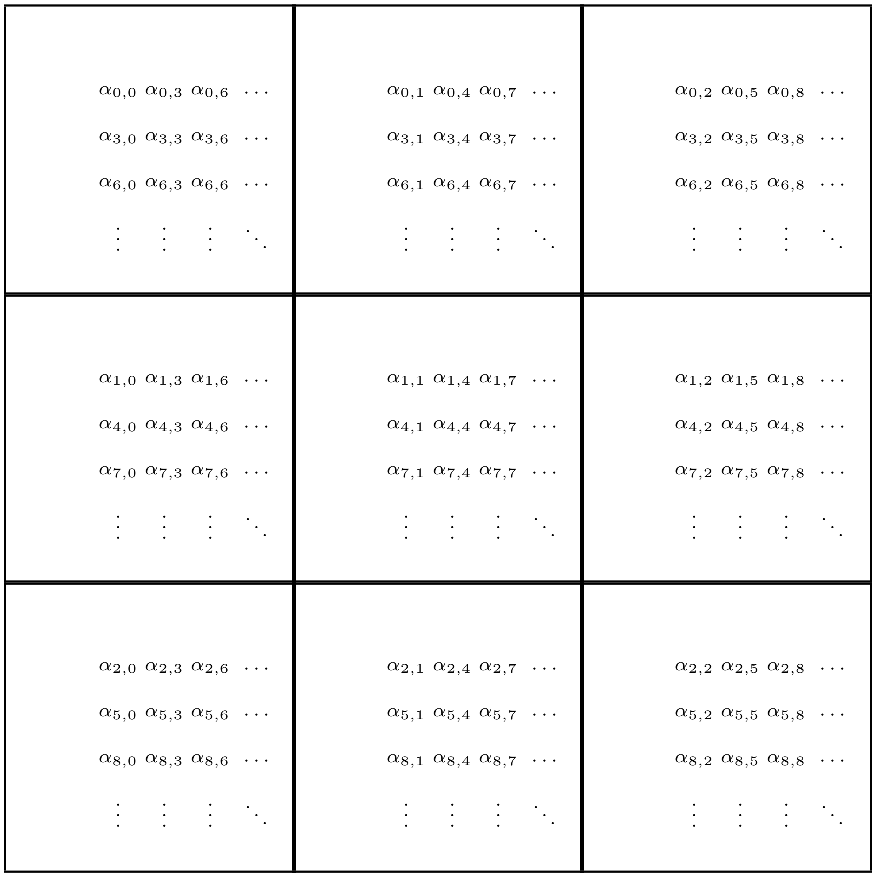
The key is to locally call a matrix-matrix multiply rather than a rank-1 update. For this, observe that

$$C = \left(\begin{array}{c|c|c|c} A_0 & A_1 & \ldots & A_{K-1} \end{array} \right) \left(\begin{array}{c} B_0 \\ B_1 \\ \vdots \\ B_{K-1} \end{array} \right) = ((\cdots((C + A_0 B_0) + A_1 B_1) + \cdots) + A_{K-1} B_{K-1})$$

Now, I will describe how to morph `ParallelRank1` into `ParallelRankK`, which will implement $C = C + A_k B_k$, where A_k has b_k columns and B_k has b_k rows.

2.1 Calling sequence

Please make the calling sequence for this new routine



$$\begin{array}{cccc|cccc|cccc}
\alpha_{0,0} & \alpha_{0,3} & \alpha_{0,6} & \cdots & \alpha_{0,1} & \alpha_{0,4} & \alpha_{0,7} & \cdots & \alpha_{0,2} & \alpha_{0,5} & \alpha_{0,8} & \cdots \\
\alpha_{3,0} & \alpha_{3,3} & \alpha_{3,6} & \cdots & \alpha_{3,1} & \alpha_{3,4} & \alpha_{3,7} & \cdots & \alpha_{3,2} & \alpha_{3,5} & \alpha_{3,8} & \cdots \\
\alpha_{6,0} & \alpha_{6,3} & \alpha_{6,6} & \cdots & \alpha_{6,1} & \alpha_{6,4} & \alpha_{6,7} & \cdots & \alpha_{6,2} & \alpha_{6,5} & \alpha_{6,8} & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\
\hline
\alpha_{1,0} & \alpha_{1,3} & \alpha_{1,6} & \cdots & \alpha_{1,1} & \alpha_{1,4} & \alpha_{1,7} & \cdots & \alpha_{1,2} & \alpha_{1,5} & \alpha_{1,8} & \cdots \\
\alpha_{4,0} & \alpha_{4,3} & \alpha_{4,6} & \cdots & \alpha_{4,1} & \alpha_{4,4} & \alpha_{4,7} & \cdots & \alpha_{4,2} & \alpha_{4,5} & \alpha_{4,8} & \cdots \\
\alpha_{7,0} & \alpha_{7,3} & \alpha_{7,6} & \cdots & \alpha_{7,1} & \alpha_{7,4} & \alpha_{7,7} & \cdots & \alpha_{7,2} & \alpha_{7,5} & \alpha_{7,8} & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \\
\hline
\alpha_{2,0} & \alpha_{2,3} & \alpha_{2,6} & \cdots & \alpha_{2,1} & \alpha_{2,4} & \alpha_{2,7} & \cdots & \alpha_{2,2} & \alpha_{2,5} & \alpha_{2,8} & \cdots \\
\alpha_{5,0} & \alpha_{5,3} & \alpha_{5,6} & \cdots & \alpha_{5,1} & \alpha_{5,4} & \alpha_{5,7} & \cdots & \alpha_{5,2} & \alpha_{5,5} & \alpha_{5,8} & \cdots \\
\alpha_{8,0} & \alpha_{8,3} & \alpha_{8,6} & \cdots & \alpha_{8,1} & \alpha_{8,4} & \alpha_{8,7} & \cdots & \alpha_{8,2} & \alpha_{8,5} & \alpha_{8,8} & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Figure 2: Distribution of A to a 3×3 mesh of processes.

```
void ParallelRankK(
   // global matrix dimensions m, n, k
   int global_m, int global_n, int global_k,
   // the global index of the current columns of A and current rows of B
   // to be used for the rank-b_k update
   int p,
                   // the block size for this rank-k update
   int bk,
   // address of where local A, B, and C are stored, as well as their
   // local "leading dimensions"
   double *local_A, int local_ldimA,
   double *local_B, int local_ldimB,
   double *local_C, int local_ldimC,
   // communicator for the row in which this node is a member
   MPI_Comm comm_row,
   // communicator for the column in which this node is a member
   MPI_Comm comm_col )
```

(Notice the new parameter `bk`.)

Store this first implementation in file `ParallelRankK_2.c`.

$\begin{matrix}\beta_{0,0} & \beta_{0,3} & \beta_{0,6} & \cdots \\ \beta_{3,0} & \beta_{3,3} & \beta_{3,6} & \cdots \\ \beta_{6,0} & \beta_{6,3} & \beta_{6,6} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{0,1} & \beta_{0,4} & \beta_{0,7} & \cdots \\ \beta_{3,1} & \beta_{3,4} & \beta_{3,7} & \cdots \\ \beta_{6,1} & \beta_{6,4} & \beta_{6,7} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{0,2} & \beta_{0,5} & \beta_{0,8} & \cdots \\ \beta_{3,2} & \beta_{3,5} & \beta_{3,8} & \cdots \\ \beta_{6,2} & \beta_{6,5} & \beta_{6,8} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$
$\begin{matrix}\beta_{1,0} & \beta_{1,3} & \beta_{1,6} & \cdots \\ \beta_{4,0} & \beta_{4,3} & \beta_{4,6} & \cdots \\ \beta_{7,0} & \beta_{7,3} & \beta_{7,6} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{1,1} & \beta_{1,4} & \beta_{1,7} & \cdots \\ \beta_{4,1} & \beta_{4,4} & \beta_{4,7} & \cdots \\ \beta_{7,1} & \beta_{7,4} & \beta_{7,7} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{1,2} & \beta_{1,5} & \beta_{1,8} & \cdots \\ \beta_{4,2} & \beta_{4,5} & \beta_{4,8} & \cdots \\ \beta_{7,2} & \beta_{7,5} & \beta_{7,8} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$
$\begin{matrix}\beta_{2,0} & \beta_{2,3} & \beta_{2,6} & \cdots \\ \beta_{5,0} & \beta_{5,3} & \beta_{5,6} & \cdots \\ \beta_{8,0} & \beta_{8,3} & \beta_{8,6} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{2,1} & \beta_{2,4} & \beta_{2,7} & \cdots \\ \beta_{5,1} & \beta_{5,4} & \beta_{5,7} & \cdots \\ \beta_{8,1} & \beta_{8,4} & \beta_{8,7} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$	$\begin{matrix}\beta_{2,2} & \beta_{2,5} & \beta_{2,8} & \cdots \\ \beta_{5,2} & \beta_{5,5} & \beta_{5,8} & \cdots \\ \beta_{8,2} & \beta_{8,5} & \beta_{8,8} & \cdots \\ \vdots & \vdots & \vdots & \ddots\end{matrix}$

Figure 3: Distribution of B to a 3×3 mesh of processes.

2.2 From `ParallelRank1` to `ParallelRankK_2`

Work arrays First, you will want to make it so that `work_A` can hold b_k columns and `work_B` can hold b_k columns. Hint: take rows of B but store them as columns of `work_B`!!! This will make your life easier.

Communication In your first implementation, perform separate broadcasts to communicate each of the b_k columns of A and b_k rows of B by putting a loop around the broadcasts in `ParallelRank1`.

Local computation Also, put a loop around the rank-1 updates (calls to `dger_`).

2.3 From `ParallelMMult_1` to `ParallelMMult_2`

Copy `ParallelMMult_1.c` into `ParallelMMult_2.c` (but don't change the name or calling sequence). Change the routine to call `ParallelRankK`. You will have to change the loop to increment in steps of `bk`. How do you deal with the fact that in the last iteration you may not have a full set of `bk` columns in A (or rows in B)?

Change the `Makefile` and see if you get the right answer.

3 Optimization 2

3.1 From `ParallelRankK_2` to `ParallelRankK_3`

Copy from one file to a new file.

Local computation Replace the loop with calls to `dger_` with a single call to the matrix-matrix multiplication routine `dgemm_`. (An example of how to call this is in the driver. Remember: `work_B` hold the transpose of the rows of B with which you are computing!)

Change the `Makefile` and see if you get the right answer.

4 Optimization 3

4.1 From `ParallelRankK_3` to `ParallelRankK_4`

Copy from one file to a new file.

Communication Replace the loop with calls to `MPI_Bcast` with a single call to an appropriate collective communication. What else do you need to do?

Change the `Makefile` and see if you get the right answer.