To appear in Proceedings of Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering (PARA95), Lyngby, Denmark, August 22–24, 1995, Lecture Notes in Computer Science, Springer-Verlag.

Distributed General Matrix Multiply and Add for a 2D Mesh Processor Network

Bo Kågström and Mikael Rännar

Department of Computing Science, University of Umeå, S-901 87 Umeå, Sweden.
Email addresses: bokg@cs.umu.se and mr@cs.umu.se.

Abstract. A distributed algorithm with the same functionality as the single-processor level 3 BLAS operation GEMM, i.e., general matrix multiply and add, is presented. With the same functionality we mean the ability to perform GEMM operations on arbitrary subarrays of the matrices involved. The logical network is a 2D square mesh with torus connectivity. The matrices involved are distributed with non-scattered blocked data distribution. The algorithm consists of two main parts, alignment and data movement of subarrays involved in the operation and a distributed blocked matrix multiplication algorithm on (sub)matrices using only a square submesh. Our general approach makes it possible to perform GEMM operations on non-overlapping submeshes simultaneously.

1 Introduction

The level 3 Basic Linear Algebra Subprograms (BLAS) [6] were developed to exploit the memory hierarchy of modern advanced architectures, including vector and RISC-based processors and parallel computers. With optimized level 3 BLAS for different machines it is possible to design portable high performance codes for many matrix computational problems (e.g. the LAPACK library [1]). The "most basic" operation in level 3 BLAS is the general matrix multiply and add (GEMM) operation

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

where $\mathrm{op}(X)$ denotes the matrix X or its transpose operation X^T, C is $m \times n$, $\mathrm{op}(A)$ is $m \times k$ and $\mathrm{op}(B)$ is $k \times n$, and α, β are scalars.

Our objective is to design and implement a general GEMM routine for scalable distributed memory machines (DMM) with the same functionality as the single processor level 3 BLAS matrix multiply and add operation. With the same functionality we mean the ability to perform GEMM operations on arbitrary subarrays of the matrices involved. Moreover, all necessary communication in the GEMM operation should be performed implicitly (and hidden for the user).

A lot of work concerning distributed GEMM on a 2-dimensional (2D) processor mesh has been published [4, 10, 14]. The most successful approaches (with respect to performance and scalability) distribute the data blockwise over the processor grid. This can be done in several ways (pure-block, block scattered (or block cyclic), virtual 2D torus map etc). When considering a distributed

GEMM in isolation pure-block matrix mapping is optimal [3] at least concerning communication overhead. Common for these approaches is that they only can perform block matrix multiplications on subarrays that conform with the block distribution of the arrays on the processor mesh.

A library version should be portable between different architectures. This is solved by using a set of communication primitives available on several platforms, e.g. PICL [9] and PVM [8]. By using optimized uniprocessor GEMM as the computational kernel and having a well-defined interface, which makes it easy to use your favourite distributed matrix multiplication algorithm, we accomplish high performance on a wide variety of DMM architectures.

Section 2 introduces the logical processor topology and the blocked data decomposition used. In Section 3 we present the algorithm for the distributed general GEMM operation (implemented in Fortran 77 as routine `ddgemm`). Section 4 presents the interface to `ddgemm`. In Section 5 we show some measured performance results from two DMM systems. Finally, Section 6 gives a brief summary of our work.

2 Processor Topology and Data Decomposition

Logical processor network. The processor network should be a square 2D mesh with torus connectivity, or at least able to embed that topology. It must also be possible to embed square subtorus networks, which in this context means a square 2D submesh with torus connectivity. The processor nodes are referenced by their row and column indices in the 2D mesh and the (i, j) node is denoted P_{ij}. The size of the complete square 2D mesh is $\texttt{DIM} \times \texttt{DIM}$. Similarly, the size of a square 2D submesh is $\texttt{SubDIM} \times \texttt{SubDIM}$.

Matrix partitioning and mapping. The matrices A, B, and C are distributed blockwise over the 2D mesh with equal-sized, possibly rectangular, blocks. When distributing an $m \times n$ matrix A over a $p \times p$ processor mesh we partition the matrix in p row blocks and p column blocks such that

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{p1} & \cdots & A_{pp} \end{bmatrix},$$

where block A_{ij} has dimension $m_i \times n_j$. For all blocks, except possibly the last block row and block column, $m_i = \lceil m/p \rceil$ and $n_j = \lceil n/p \rceil$. If p is not an even multiple of m and n, then the blocks in the last block row and the last block column will have smaller dimensions. Initially, node P_{ij} holds the blocks A_{ij}, B_{ij} and C_{ij}.

With arbitrary subarrays we mean any consecutive rows and columns of the full matrix, no matter which block boundaries we have due to the data distribution over the processor mesh. Since we allow arbitrary subarrays that cross the block partitioning, different processors may store blocks of subarrays of different sizes, which has to be dealt with separately. Notice, when it is clear from the context, A_{ij} may refer to a block of a subarray A.

3 Algorithm for a Distributed General GEMM

We present an algorithm for computing $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$, where $\mathrm{op}(X) = X$ or X^T, on any subarrays of the matrices A, B and C with a single call to a subroutine `ddgemm`, without having to think about on which nodes different parts of the subarrays are stored. The acronym stands for `d`istributed, `d`ouble precision real, `ge`neral `m`atrix, `m`ultiply and add, i.e., `ddgemm` implements a distributed general GEMM operation.

A novelty of `ddgemm` is that it only uses a (square) 2D submesh large enough to hold the different subarrays. This makes it possible to have several distributed GEMM operations going on in the complete 2D mesh, which, e.g. is of interest in a GEMM-based approach [11, 12]. As in the single node `dgemm`, C is the only subarray changed, and since copies of subarrays A and B are used in data transfers between nodes it is possible to use one of the subarrays A and B as "C", e.g. $A = A + B \cdot A$.

The distributed GEMM algorithm consists of the following four major steps:

1. Determine the size (`SubDIM`) and the placement of the subtorus to be used in the distributed block algorithm.
2. Adjust subarrays A and B such that the local GEMM operations of the distributed algorithm are well-defined.
3. Move subarrays A and B into subtorus using the complete network.
4. Choose your favourite distributed algorithm for a general matrix multiply and add operation.

These four steps are captured in separate subroutines that are briefly described below. For more details see [13].

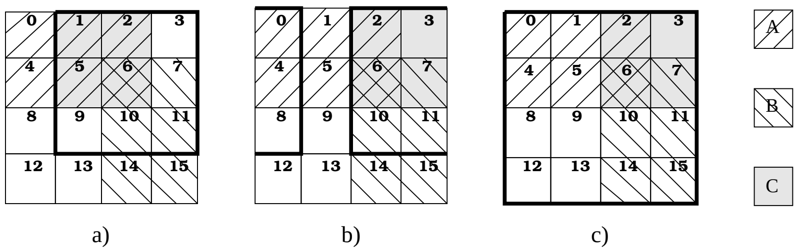

Fig. 1. Illustration of the expand operation. (Subtorus inside the thick lines.)

`expand`: The routine sets `SubDIM` to the largest number of processors storing rows and columns of the subarrays involved in the GEMM operation. For example, in Figure 1 the subarrays A and B are stored on a 2×3 and 3×2 mesh, respectively, so $\texttt{SubDIM} = 3$. The starting processor of the subtorus is set to be the processor storing the $(1, 1)$ block of subarray C. In Figure 1 we show how the submesh can be selected. All three cases have the same sizes of the subarrays involved in the GEMM operation, but in a) and b) with different starting indices of subarray C. In Figure 1 c) the whole processor mesh is selected as the subtorus. Case c) illustrates that `expand` can be changed to utilize facts about the processor network, e.g. that you always want to use the complete 2D mesh.

`reshape`: The routine adjusts a subarray so that given the subarray C's placement and size, all local multiplications in the distributed multiplication algorithm are well-defined. The reshape operation is performed on both A and B in the GEMM operation, while the subarray C is never touched. Each call performs zero, one or two shift operations (send/receive some rows/columns to/from a nearest neighbour) such that the new *local* starting row of subarray A is the same as the local starting row of subarray C and the new local starting column of subarray B is the same as the local starting column of subarray C.

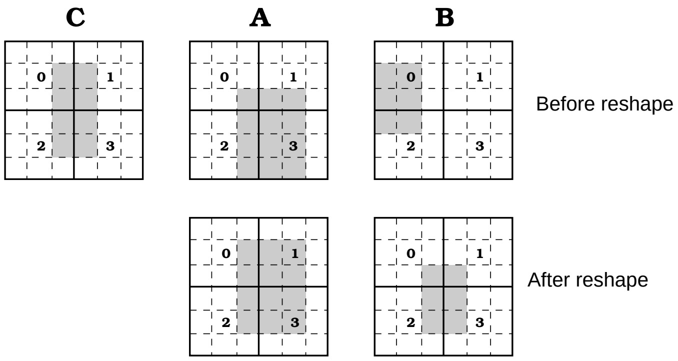

Fig. 2. Illustration of a reshape operation.

In Figure 2 we illustrate the reshape operation on three 6×6 matrices distributed on a 2×2 processor mesh. We wish to compute $C(2:5, 3:4) = A(3:6, 3:5) \cdot B(3:5, 1:2) + C(2:5, 3:4)$, which involves the gray areas in the figure. After we have applied `reshape` to subarrays A and B, all local matrix multiplications in the distributed algorithm are legal. For instance, node 0 will now multiply a 2×1 matrix with a 1×1, and add the result to a 2×1 matrix, which is a well-defined GEMM operation.

If the subarrays A and B do not overlap then the alignment of the subarrays are done in parallel, but nodes that store parts of both subarrays will have to do the reshape operation sequentially.

`move`: Since we only use the subtorus to perform the distributed matrix multiplication we need to move the subarrays A and B inside the subtorus, which is accomplished with the routine `move`. The $(1,1)$ blocks are moved to the node storing the $(1,1)$ block of subarray C, and so on for all blocks in the subarrays. The subarray C is never moved, because the result of the operation is stored in C, and we would have to move C back again. In some situations it could be more efficient to move C too.

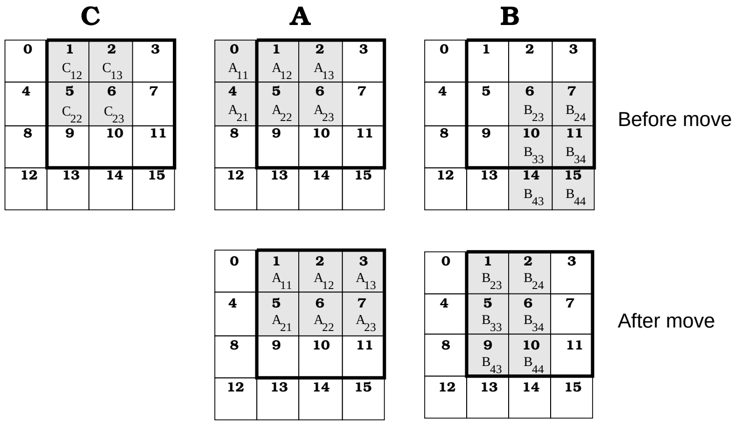

Fig. 3. Illustration of the move operation.

The shortest way is chosen with respect to the torus connectivity. The subarray is moved vertically first, and then horizontally, with only nearest neighbour communication. If the subarrays A and B originally have some nonoverlapping parts, then some of the sends and receives are done in parallel. In the end of the moves, e.g. inside the subtorus, the communication must be done serially. In Figure 3 we show an example of how the distribution may look like before and after the calls to `move`. After the move operation, all nodes outside of the subtorus return from `ddgemm` and are free to do other work (continue in the node program).

`distblockalg`: The routine performs the actual GEMM operation, and the "input" to the routine is a square subtorus (represented by a starting node and size), sizes of the local blocks of the subarrays, neighbours, and row and column indices of the node in the subtorus. Given all this information it is possible to implement different distributed algorithms for computing $C = \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$, utilizing an optimized single node GEMM routine. We have implemented blocked variants of two algorithms [7], [2, 5].

The Block Torus Matrix Multiply (BTMM) algorithm is a modified blocked variant of Cannon's algorithm where each node stores a (rectangular) block of each matrix A, B, and C [2, 5]. Initially, A is skewed rowwise and B is skewed columnwise such that P_{ij} holds the blocks A_{ik} and B_{kj}, where $k = (i + j - 2) \bmod \texttt{SubDIM} + 1$. From now on the communication between nodes is minimized since A and B need to be shifted circularly only one position (east and south, respectively) for each of the `SubDIM` iterations of the block matrix multiplication.

In the Broadcast Multiply and Roll (BMR) algorithm [7] no initial skewing of the matrices is needed. Instead a broadcast operation of a block of subarray A along the rows of the submesh is performed in each iteration, followed by a local GEMM operation and a roll operation that shifts subarray B circularly one row. The broadcast can be implemented using a pipelined algorithm, with only nearest neighbour communication.

The original algorithms are used on square matrices, evenly distributed on the square processor mesh. Here the subarrays need not be square, and the local blocks of subarrays can even be of size zero on some nodes. We handle the zero matrix problem by letting all nodes in the subtorus take part in the algorithms as if they all had parts of all subarrays. If a node does not have a block of subarray A or B to send then the node sends an empty message, and the receiving node then knows that it does not have to do the local multiplication in this step of the algorithm. In the next step however it may receive a full block. It is only the nodes storing part of subarray C and in this step have nonempty blocks of subarrays A and B that do local GEMM operations. All other nodes just take part in data movements. For example, in Figure 3, nodes 1, 2, 5, and 6 do local multiplications, and all nodes inside the thick line (defining the subtorus) take part in data movement operations, e.g. row-broadcast of A in the BMR algorithm.

Communication overhead. The routines `reshape`, `move` and `distblockalg` contribute to the communication overhead of `ddgemm`. Notice that there is no communication at all in the routine `expand`. In `reshape` and `move` the data movements do not only depend on the block sizes, but also on starting indices of the subarrays involved in the GEMM operation and whether `BLOCK` is `.TRUE.` or `.FALSE.` (see Section 4). For example, if `BLOCK` = `.TRUE.`, `reshape` is not called. Otherwise, `reshape` contributes with communication overhead for each subarray that has to be adjusted in row and/or column directions. An analysis of the communication overhead of `reshape` and `move` is presented in [13]. The communication cost of `distblockalg` is of course dependent on which distributed algorithm that is used. For example, the total communication of the BMR algorithm involves $\texttt{SubDim}^2$ send and receive operations of full blocks.

4 Interface to ddgemm

The calling sequence of `ddgemm` is extended with some parameters that define the data layout and the subarrays involved in the operation.

```
ddgemm(TransA, TransB, m, n, k, alpha, A, LDA, B, LDB, beta, C, LDC,
       Ablksz, Bblksz, Cblksz, BLOCK, StartA, StartB, StartC, TmpM,
       SIZE, Info)
```

The arguments of `ddgemm` on the first line are the same as for the level 3 BLAS `dgemm` routine [6] (see argument `BLOCK` for the exceptions on `m`, `n`, and `k`). The additional arguments are described below, where all except `Info` are input arguments. Examples of the use of `ddgemm` can be found in [13].

- `Xblksz`, `integer(2)` Row and column block sizes of matrix `X`.
- `BLOCK`, `logical` Set to `.TRUE.` if call made with block indices, otherwise `.FALSE.`. Normally, `BLOCK` is set to `.FALSE.` only if any of the subarrays involved in the distributed GEMM operation cross the block partitioning boundaries.
  The six parameters `m`, `n`, `k`, `StartA`, `StartB`, `StartC` are all controlled by `BLOCK`, e.g. if `BLOCK` is `.TRUE.` and `m` = `n` = 1, then the size of subarray C is `Cblksz(1)` $\times$ `Cblksz(2)`, but if `BLOCK` is `.FALSE.` then the size of subarray C is 1×1.
- `StartX`, `integer(2)` Starting row and column indices of subarray `X`. The values of `StartX` is controlled by `BLOCK`. If `BLOCK` = `.TRUE.`, `StartX` holds global block indices of subarray `X`. If `BLOCK` = `.FALSE.`, `StartX` holds global elementwise indices of subarray `X`.
- `TmpM`, `double precision(*)` Workspace of size `SIZE`.
- `SIZE`, `integer` The size of the workspace in double precision words. At least large enough to hold copies of the local blocks of A and B, plus workspace for the larger of the two, i.e., if $Asize$ = `Ablksz(1)` $\cdot$ `Ablksz(2)` and $Bsize$ = `Bblksz(1)` $\cdot$ `Bblksz(2)` then `SIZE` must be at least $Asize + Bsize + \max(Asize, Bsize)$.
- `Info`, `integer` Diagnostic argument that indicates the success or failure of the computation.

A template of the node program for `ddgemm` can be found in [13]. To be able to have several distributed GEMM operations going on simultaneously in the 2D mesh we have to distinguish messages from different calls to `ddgemm`. We use an internal counter that is updated every time a call is made. To keep this information correct we have to impose that all nodes make a call to `ddgemm`. Some of these nodes may only perform data movements, and some perform both data movements and arithmetic computations, while some nodes may not be involved in any part of the operation and they can return immediately from `ddgemm`, and may, e.g. take part in another distributed GEMM operation.

5 Some Performance Results

We have tested ddgemm on two DMM computers, Intel iPSC/2 (Umeå) and Intel Paragon (Bergen). Here we show results from two computational experiments. First we let the subarrays start on the same block index and vary the sizes and "shapes" of the subarrays A, B, and C. In the second experiment we fix the sizes of the subarrays and the placement of the subarrays A and C, while varying the starting index of subarray B.

We measure the parallel speedup, S_p and the efficiencies, E_{p_c} and E_p as

$$S_p = \frac{\text{time of optimized single node } \texttt{dgemm}}{\text{time of } \texttt{ddgemm} \text{ on } p \text{ nodes}} \quad , \quad E_{p_c} = \frac{S_p}{p_c} \quad , \quad E_p = \frac{S_p}{p} \quad , \qquad (1)$$

where p_c is the number of nodes storing a block of subarray C and p is the number of nodes in the submesh ($\texttt{SubDIM} \cdot \texttt{SubDIM}$).

Maximal parallel speedup. Since only nodes storing parts of C are involved in the arithmetic computations of the distributed GEMM operation the maximum value of S_p is limited to p_c. For example, in Figure 3 we are using 9 nodes in the submesh, but there are only 4 nodes storing subarray C. This explains why we are measuring E_{p_c}. Notice that $E_p = p_c/p \cdot E_{p_c} \leq E_{p_c}$, with equality if $p_c = p$.

Experimental speedup. We have tested ddgemm on a number of different shaped subarrays, e.g. A square, B and C rectangular. Some of the results for the Intel iPSC/2 and the Intel Paragon are presented in tables 1 and 2. The GEMM operation is done on 16 nodes (a 4×4 mesh). In both cases we have used the largest matrix possible that fit into the memory of one processor of the parallel machine, 192×192 on the iPSC/2 and 920×920 on the Paragon. At least one of m, n or k is as large as possible, which means that the dimension of the torus involved in the multiplication is 4, i.e., the whole 4×4 processor mesh is always used.

Since all subarrays start on the same global index, the reshape and move operations do not need to be performed. Therefore this experiment is a test on how close to the maximal parallel speedup p_c we can reach. The first column of tables 1 and 2 displays the sizes of the involved subarrays. The second and third columns display the corresponding times (measured in milliseconds (ms)) for the optimized single node dgemm and ddgemm executing on 16 nodes, respectively.

We can see from Table 1 that E_{p_c} increases with k, on the iPSC/2. If $k = 192$ the efficiency stays around 98%, but if $k = 48$, it drops to about 87%. This degradation is correlated to the number of blocks in "k-direction", which is equal to the number of local GEMM operations done by each node storing a block of subarray C. The more local multiplications, the better performance, because more communication can be hidden during arithmetic work. For example, if $k = 192$ all C-nodes make 4 local calls to dgemm, but if $k = 48$, there is only 1 call. In both cases the number of send and receive operations are the same.

Table 1. Performance results on Intel iPSC/2, using 16 nodes (4×4 mesh)

Matrix sizes m	n	k	dgemm 1 node	ddgemm 16 nodes	S_p	p_c	E_{p_c} (%)
48	48	192	2833	2847	0.995	1	100
48	192	192	11332	2887	3.93	4	98
48	192	96	5448	1485	3.67	4	92
192	48	48	2688	769	3.50	4	87
192	192	48	10745	786	13.67	16	85
96	192	48	5365	767	6.99	8	87
192	48	192	11238	2880	3.90	4	98
192	192	192	44956	2930	15.34	16	96
96	192	96	10974	1488	7.38	8	92
96	48	192	5632	2864	1.97	2	98
96	96	192	11261	2885	3.90	4	98
48	96	192	5665	2858	1.98	2	99
48	192	48	2723	766	3.55	4	89

The results for the Intel Paragon are similar, as shown in Table 2, but with lower efficiencies which is mainly due to the higher communication/computation ratio of the Intel Paragon. The efficiency should increase if we use larger subarrays and the same proportions between m, n and k.

Timing with move. To see how `ddgemm` behaves if the subarrays do not start on the same node, we have done some test runs where the subarrays A and C are fixed (they always start in the same place), but subarray B moves around to different starting points. The submesh is 4×4 and has its upper left corner on the node that stores the $(1, 1)$ block of subarray C. All multiplications are done on full blocks. The experiment is "worst case" in the sense that the subarrays A, B and C are non-overlapping and the subarrays A and B (after position 4) are outside the submesh (see Figure 4).

Table 3 shows the results for Intel Paragon. The first column in the table gives references to the numbers in Figure 4, where we illustrate how the subarray B is placed and moved around in the processor mesh. All matrices are 3680×3680, and the multiplications are done on subarrays with $m = k = 1840, n = 460$.

In this test we have used the same dimensions as in example number 7 in the experimental speedup test. The difference is that now we have an 8×8 processor mesh, instead of a 4×4 mesh, and larger global matrices (but the local sizes are the same). The GEMM operation is still done in a 4×4 submesh, starting at node P_{54}.

As we can see in Table 3 the difference is only a few percent even if the distance from B to the submesh increases. Sometimes the execution time even becomes smaller as the distance increases, e.g. between positions 7 and 8 in

Table 2. Performance results on Intel Paragon, using 16 nodes (4×4 mesh).

Matrix sizes m	n	k	dgemm 1 node	ddgemm 16 nodes	S_p	p_c	E_{p_c} (%)
230	230	920	2231	2601	0.86	1	86
230	920	920	8575	2697	3.18	4	79
230	920	460	4378	1546	2.83	4	71
920	230	230	2275	936	2.43	4	61
920	920	230	8730	936	9.33	16	58
460	920	230	4461	847	5.27	8	66
920	230	920	8717	2590	3.37	4	84
920	920	920	34187	2737	12.49	16	78
460	920	460	8751	1538	5.69	8	71
460	230	920	4448	2661	1.67	2	84
460	460	920	8740	2664	3.28	4	82
230	460	920	4343	2694	1.61	2	81
230	920	230	2252	895	2.52	4	63

Table 3. Times with `move` on Paragon, using 64 nodes (logically a 8×8 mesh).

No.	Time (ms)	No.	Time (ms)	No.	Time (ms)
1	20091	7	20762	13	20812
2	20250	8	20289	14	20726
3	20251	9	20260	15	20597
4	20561	10	20580	16	20842
5	20386	11	20642	17	20851
6	20728	12	20979		

Table 3. The reason is that most of the move of B can be done in parallel with the move of A, as there are different nodes involved. If we compare the results in Table 3 with the "ideal" case (18266 ms), when the subarrays A, B and C start on the same processor, using 16 nodes, then it costs 10–15 % of the time to use the move operation. We somehow have to move the subarrays around and `ddgemm` does it implicitly.

6 Conclusions

We have developed a distributed general matrix multiply and add routine `ddgemm` in Fortran 77 for double precision real data that is portable, easy to use and can operate on subarrays as well as on full matrices. Instead of having to explicitly redistribute the subarrays A, B, and C, the user just specify indices that specify the subarray operation in almost the same way as in an ordinary single processor

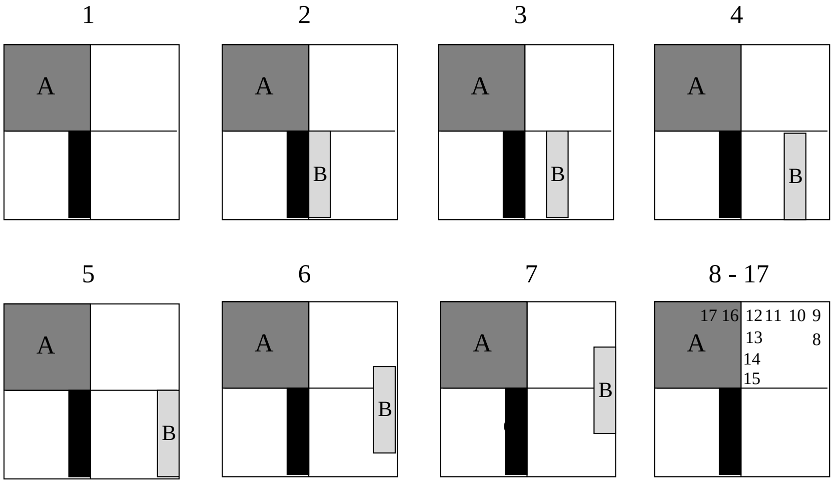

Fig. 4. Subarrays A, B and C on a 8×8 processor mesh, where the placement of B is varied to 17 positions with different starting indices. Each square (e.g. the area A) is a 4×4 mesh.

`dgemm` routine. Another novelty of `ddgemm` is that it only uses a square 2D submesh large enough to store the different subarrays, which makes it possible to perform several GEMM operations (or other computations) on non-overlapping submeshes simultaneously. Computational experiments show that our approach is effective.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.
2. V. Cherkassky and R. Smith. Efficient mapping and implementation of matrix algorithms on a hypercube. *Journal of Supercomputing*, 2(1):7–27, 1988.
3. J. Choi, J. J. Dongarra, and D. W. Walker. Level 3 BLAS for distributed memory concurrent computers. In *CNRS–NSF Workshop on Environments and Tools for Parallel Scientific Computing (Saint Hilaire du Touvet, France, September 7–8, 1992)*. Elsevier Science Publishers, 1992.
4. J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. Technical Report ORNL/TM–12252, Oak Ridge National Laboratory, Oak Ridge, TN, April 1993.
5. E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal of Computing*, 10(4):657–675, November 1981.
6. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 18(1):1–17, 1990.

7. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice–Hall, 1988.
8. G. A. Geist, A. Beguelin, Dongarra J. J., R. Manchek, and V. Sunderam. PVM 3.0 User's Guide and Reference Manual. Technical Report ORNL/TM–12187, Oak Ridge National Laboratory, Oak Ridge, TN, February 1993.
9. G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL: A portable instrumented communication library. Technical Report ORNL/TM–11616, Oak Ridge National Laboratory, Oak Ridge, TN, September 1990.
10. S. Huss-Lederman, E. M. Jacobson, and G. Tsao, A. Zhang. Matrix multiplication on the Intel Touchstone Delta. Technical Report SRC–TR–93–101 (Revised), Supercomputing Research Center, Bowie, MD, February 1994.
11. B. Kågström, P. Ling, and C. Van Loan. High Performance GEMM–Based Level 3 BLAS: Sample Routines for Double Precision Real Data. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, pages 269–281, Amsterdam, 1991. North–Holland.
12. B. Kågström, P. Ling, and C. Van Loan. Portable High Performance GEMM–Based Level 3 BLAS. In Richard F. et al Sincovec, editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM Publications.
13. M. Rännar. A Distributed, Portable and General GEMM Operation for a 2D Mesh Processor Network. Report UMINF–95.xx, Department of Computing Science, Umeå University, S–901 87 Umeå, Sweden, 1995.
14. R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report UT CS-95-286, LAPACK Working Note # 96, University of Tennessee, 1995.