# A Variable Group Block Distribution Strategy for Dense Factorizations on Networks of Heterogeneous Computers

Alexey Lastovetsky, Ravi Reddy

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

{alexey.lastovetsky, manumachu.reddy}@ucd.ie

**Abstract.** In this paper, we present a static data distribution strategy called Variable Group Block distribution to optimize the execution of factorization of a dense matrix on a network of heterogeneous computers. The distribution is based on a functional performance model of computers, which tries to capture different aspects of heterogeneity of the computers including the (multi-level) memory structure and paging effects.

## 1 Introduction

The paper presents a static data distribution strategy called Variable Group Block distribution to optimize the execution of factorization of a dense matrix on a network of heterogeneous computers. The Variable Group Block distribution strategy is a modification of Group Block distribution strategy, which was proposed in [1] for 1D parallel Cholesky factorization, developed into a more general 2D distribution strategy in [2] and applied to 1D LU factorization in [3], [4].

The Group Block distribution strategy is based on the performance model, which represents the speed of each processor by a constant positive number and computations are distributed amongst the processors such that their volume is proportional to this speed of the processor. However the single number model is efficient only if the relative speeds of the processors involved in the execution of the application are a constant function of the size of the problem and can be approximated by a single number. This is true mainly for homogeneous distributed memory systems where:

- The processors have almost the same size at each level of their memory hierarchies, and
- Each computational task assigned to a processor fits in its main memory.

But the model becomes inefficient in the following cases:

- The processors have significantly different memory structure with different sizes of memory at each level of memory hierarchy. Therefore, beginning from some problem size, the same task will still fit into the main memory of some processors and stop fitting into the main memory of others, causing the paging and visible degradation of the speed of these processors. This means that their relative speed will start significantly changing in favor of non-paging processors as soon as the problem size exceeds the critical value.

– Even if the processors of different architectures have almost the same size at each level of the memory hierarchy, they may employ different paging algorithms resulting in different levels of speed degradation for the task of the same size, which again means the change of their relative speed as the problem size exceeds the threshold causing the paging.

Thus considering the effects of processor heterogeneity, memory heterogeneity, and the effects of paging significantly complicates the design of algorithms distributing computations in proportion with the relative speed of heterogeneous processors. One approach to this problem is to just avoid the paging as it is normally done in the case of parallel computing on homogeneous multi-processors. However avoiding paging in local and global heterogeneous networks may not make sense because in such networks it is likely to have one processor running in the presence of paging faster than other processors without paging. It is even more difficult to avoid paging in the case of distributed computing on global networks. There may not be a server available to solve the task of the size you need without paging.

Therefore, to achieve acceptable accuracy of distribution of computations across heterogeneous processors in the possible presence of paging, a more realistic performance model of a set of heterogeneous processors is needed. In [5], we suggested a functional performance model of computers that integrates some of the essential features underlying applications run on general-purpose common heterogeneous networks, such as the processor heterogeneity in terms of the speeds of the processors, the memory heterogeneity in terms of the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy, and the effects of paging. Under this model, the speed of each computer is represented by a continuous and relatively smooth function of problem size.

The Variable Group Block distribution strategy presented in this paper uses this functional performance model to optimize the execution of factorization of a dense square matrix on a network of heterogeneous computers.

The functional model does not take into account the effects on the performance of the processor caused by several users running heavy computational tasks simultaneously. It supposes only one user running heavy computational tasks and multiple users performing routine computations and communications, which are not heavy like email clients, browsers, audio applications, text editors etc.

The rest of the paper is organized as follows. In the next section, we present the Variable Group Block distribution strategy. We then show experimental results on a local network of heterogeneous computers to demonstrate the efficiency of the Variable Group Block Distribution strategy over the Group Block Distribution Strategy.

## 2 Variable Group Block Distribution

Before we present our Variable Group Block distribution strategy, we briefly explain the LU Factorization algorithm of a dense (**n**×**b**)×(**n**×**b**) square matrix *A*, one step of which is shown in Figure 1. **n** is the number of blocks of size **b**×**b** [6], [7]. On a homogeneous **p**-processor linear array, a CYCLIC(**b**) distribution of columns is used to distribute the matrix *A*. The cyclic distribution would assign columns of blocks with

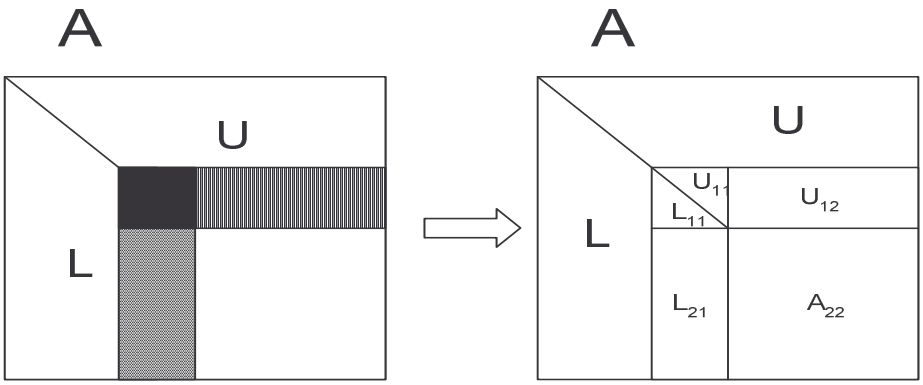


**Fig. 1.** One step of the LU factorization algorithm of a dense square matrix (**n**×**b**)×(**n**×**b**).

numbers 0,1,2,...,**n**-1 to processors 0,1,2,...,**p**-1,0,1,2,...,**p**-1,0,..., respectively, for a **p**-processor linear array (**n**≫**p**), until all **n** columns of blocks are assigned. At each step of the algorithm, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining blocks. At the next step, the next column of **b**×**b** blocks becomes the pivot panel, and the computation progresses. Figure 1 shows how the column panel, $L_{11}$ and $L_{21}$, and the row panel, $U_{11}$ and $U_{12}$, are computed and how the trailing submatrix $A_{22}$ is updated. Because the largest fraction of the work takes place in the update of $A_{22}$, therefore, to obtain maximum parallelism all processors should participate in the updating. Since $A_{22}$ reduces in size as the computation progresses, a cyclic distribution is used to ensure that at any stage $A_{22}$ is evenly distributed over all processors, thus obtaining a balanced load.

Two load balancing algorithms, namely, Group Block algorithm and Dynamic Programming algorithm [7] have been proposed to obtain optimal static distribution over **p** heterogeneous processors arranged in a linear array. The Group Block distribution partitions the matrix into groups (or *generalized blocks* in terms of [2]), all of which have the same number of blocks. The number of blocks per group (size of the group) and the distribution of the blocks in the group amongst the processors are fixed and are determined based on speeds of the processors, which are represented by a single constant number. Same is the case with Dynamic Programming distribution except that the distribution of the blocks in the group amongst the processors is determined based on dynamic programming algorithm.

We propose a static distribution strategy called Variable Group Block distribution, which is a modification of the Group Block algorithm. It uses the functional model where absolute speed of the processor is represented by a function of a size of the problem. Since the Variable Group Block distribution uses the functional model where absolute speed of the processor is represented by a function of a size of the problem, the distribution uses absolute speeds at each step of the LU factorization that are based on the size of the problem solved at that step. That is at each step, the number of blocks per group and the distribution of the blocks in the group amongst the processors are

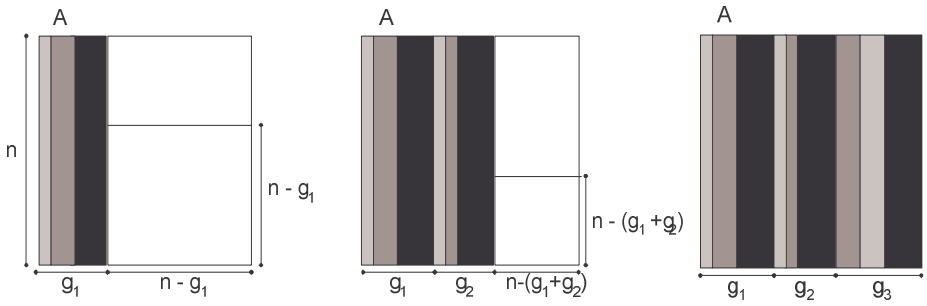


**Fig. 2.** The matrix *A* is partitioned using Variable Group Block distribution. The size of the matrix is shown in blocks of size **b**×**b**. This figure illustrates the distribution for **n**=18,**b**=32,**p**=3. The distribution inside groups $G_1$, $G_2$, and $G_3$ are {2,1,1,0,0,0}, {2,1,0,0,0}, and {2,2,1,1,0,0,0}. At each step of the distribution, the absolute speed of the processor is obtained based on the update of the trailing matrix. Since the Variable Group Block distribution uses the functional model where the absolute speed of the processor is represented by a function of the problem size, the distribution uses absolute speeds at each step that are based on the size of the problem solved at that step.

determined based on absolute speeds of the processors given by the functional model, which are based on solving the problem size at that step. Thus it takes into account the effects of (multi-level) memory structure and paging.

Figure 2 illustrates the Variable Group Block distribution of a dense square (**n**×**b**)×(**n**×**b**) matrix *A* over **p** heterogeneous processors. The Variable Group Block distribution is a static data distribution that vertically partitions the matrix into **m** groups of blocks of size **b** whose column sizes are $\mathbf{g}_1,\mathbf{g}_2,\ldots,\mathbf{g}_\mathbf{m}$ as shown in Figure 2.

The groups are non-square matrices of sizes (**n**×**b**)×(**g**$_1$×**b**), (**n**×**b**)×(**g**$_2$×**b**), ..., (**n**×**b**)×(**g**$_\mathbf{m}$×**b**) respectively. The steps involved in the distribution are:

1. The size $\mathbf{g}_1$ of the first group $G_1$ of blocks is calculated as follows:
   - Using the data partitioning algorithm [5], we obtain an optimal distribution of matrix *A* such that the number of blocks assigned to each processor is proportional to the speed of the processor. The optimal distribution derived is given by ($x_i$, $s_i$) ($0 \leq i \leq \mathbf{p}-1$), where $x_i$ is the size of the subproblem such that $\sum_{i=0}^{\mathbf{p}-1} x_i=\mathbf{n}^2$ and $s_i$ is the absolute speed of the processor used to compute the subproblem $x_i$ for processor i. Calculate the load index $l_i = \frac{s_i}{\sum_{k=0}^{\mathbf{p}-1} s_k}$ ($0 \leq i \leq \mathbf{p}-1$).
   - The size of the group $\mathbf{g}_1$ is equal to $\lfloor 1/min(l_i) \rfloor$ ($0 \leq i \leq \mathbf{p}-1$). If $\mathbf{g}_1/\mathbf{p} < 2$, then $\mathbf{g}_1=\lfloor 2/min(l_i) \rfloor$. This condition is imposed to ensure there is sufficient number of blocks in the group.
   - This group $G_1$ is now partitioned such that the number of blocks $g_{1,i}$ is proportional to the speeds of the processors $s_i$ where $\sum_{i=0}^{\mathbf{p}-1} g_{1,i}=\mathbf{g}_1$ ($0 \leq i \leq \mathbf{p}-1$).
2. To calculate the size $\mathbf{g}_2$ of the second group, we repeat step 1 for the number of blocks equal to $(\mathbf{n}\text{-}\mathbf{g}_1)^2$ in matrix *A*. This is represented by the sub-matrix $\mathbf{A}_{\mathbf{n}\text{-}\mathbf{g}_1,\mathbf{n}\text{-}\mathbf{g}_1}$

**Table 1.** Specifications of the twelve computers. Paging is the size of the matrix beyond which point paging started happening.

| Machine Name | Architecture | cpu MHz | Total Main Memory (kBytes) | Available Main Memory (kBytes) | Cache (kBytes) | Paging (LU) |
|---|---|---|---|---|---|---|
| X1 | Linux 2.4.20-20.9 i686 Intel Pentium III | 997 | 513304 | 363264 | 256 | 6000 |
| X2 | Linux 2.4.18-3 i686 Intel Pentium III | 997 | 254576 | 65692 | 256 | 5000 |
| X3 | Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM) | 2783 | 7933500 | 2221436 | 512 | 11000 |
| X4 | Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM) | 2783 | 7933500 | 3073628 | 512 | 11000 |
| X5 | Linux 2.4.18-10smp Intel(R) XEON(TM) | 1977 | 1030508 | 415904 | 512 | 8500 |
| X6 | Linux 2.4.18-10smp Intel(R) XEON(TM) | 1977 | 1030508 | 364120 | 512 | 8500 |
| X7 | Linux 2.4.18-10smp Intel(R) XEON(TM) | 1977 | 1030508 | 215752 | 512 | 8000 |
| X8 | Linux 2.4.18-10smp Intel(R) XEON(TM) | 1977 | 1030508 | 134400 | 512 | 6500 |
| X9 | Linux 2.4.18-10smp Intel(R) XEON(TM) | 1977 | 1030508 | 134400 | 512 | 6500 |
| X10 | SunOS 5.8 sun4u sparc SUNW,Ultra-5_10 | 440 | 524288 | 409600 | 2048 | 5000 |
| X11 | SunOS 5.8 sun4u sparc SUNW,Ultra-5_10 | 440 | 524288 | 418816 | 2048 | 5000 |
| X12 | SunOS 5.8 sun4u sparc SUNW,Ultra-5_10 | 440 | 524288 | 395264 | 2048 | 5000 |

shown in Figure 2. We recursively apply this procedure until we have fully vertically partitioned the matrix *A*.

3. For algorithms such as LU Factorization, only blocks below the pivot are updated. The global load balancing is guaranteed by the distribution in groups; however, for the group that holds the pivot it is not possible to balance the workload due to the lack of data. Therefore it is possible to reduce the processing time if the last blocks in each group are assigned to fastest processors, that is when there is not enough data to balance the workload then it should be the fastest processors doing the work. That is in each group, processors are reordered to start from the slowest processors to the fastest processors for load balance purposes.

In LU Factorization, the size of the matrix shrinks as the computation goes on. This means that the size of the problem to be solved shrinks with each step. Consider the first step. After the factorization of the first block of **b** columns, there remain **n**-1 blocks of **b** columns to be updated. At the second step, the number of blocks of **b** columns to update is only **n**-2. Thus the speeds of the processors to be used at each step should be based on the size of the problem solved at each step, which means that for the first step, the absolute speed of the processors calculated should be based on the update of **n**-1 blocks of **b** columns and for the second step, the absolute speed of the processors calculated should be based on the update of **n**-2 blocks of **b** columns. Since the Variable Group Block distribution uses the functional model where absolute speed of the processor is represented by a function of a size of the problem, the distribution uses absolute speeds at each step that are calculated based on the size of the problem solved at that step.

For two dimensional processor grids, the Variable Group Block algorithm is applied to columns and rows independently.

## 3 Experimental Results

A small heterogeneous local network of 12 different Solaris and Linux workstations shown in Table 1 is used in the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The amount of memory, which is the difference between the main memory and free main memory shown in the tables, is used by the operating system processes and few other user application processes that perform routine computations and communications such as email clients, browsers, text editors, audio applications etc. These processes use a constant percentage of CPU.

For the parallel LU factorization application, the absolute speed of a processor must be obtained based on the execution of DGEMM routine on a dense non-square matrix of size $\mathbf{m}_1 \times \mathbf{m}_2$. The reason is that the computational cost of the application mainly falls into the update of the trailing submatrix, which is performed by this routine. Even though there are two parameters $\mathbf{m}_1$ and $\mathbf{m}_2$ representing the size of the problem, the parameter $\mathbf{m}_1$ is fixed and is equal to **n** during the application of the set partitioning algorithm [5].

To apply the set partitioning algorithm to determine the optimal data distribution for such an application, we need to extend it for problem size represented by two parameters, $\mathbf{m}_1$ and $\mathbf{m}_2$. The speed function of a processor is geometrically a surface when represented by a function of two parameters $\mathbf{s}=\mathbf{f}(\mathbf{m}_1,\mathbf{m}_2)$. However since the parameter $\mathbf{m}_1$ is fixed and is equal to **n**, the surface is reduced to a line $\mathbf{s}=\mathbf{f}(\mathbf{m}_1,\mathbf{m}_2)=\mathbf{f}(\mathbf{n},\mathbf{m}_2)$. The set partitioning algorithm can be extended here easily to obtain optimal solutions for problem spaces with two or more parameters representing the problem size. Each such problem space is reduced to a problem formulated using a geometric approach and tackled by extensions of our geometric set-partitioning algorithm. Con-sider for example the case of two parameters representing the problem size where neither of them is fixed. In this case, the speed functions of the processors are represented by surfaces. The optimal solution provided by a geometric algorithm would divide these surfaces to produce a set of rectangular partitions equal in number to the number of processors such that the number of elements in each partition (the area of the partition) is proportional to the speed of the processor.

The absolute speed of the processor in number of floating point operations per second is calculated using the formula $(2\times\mathbf{n}\times\mathbf{b}\times\mathbf{n}\times\mathbf{b})$/(execution time) where $\mathbf{n}\times\mathbf{b}$ is the size of the dense square matrix. The computer X6 exhibited the fastest speed of 130 MFlops for execution of DGEMM routine on a dense 8500×8500 matrix whereas the computer X1 exhibited the lowest speed of 19 MFlops for execution of DGEMM routine on a dense 4500×4500 matrix. The ratio $130/19 \approx 6.8$ suggests that the processor set is reasonably heterogeneous and it should also be noted that paging has not started happening at this problem size for both the computers.

We use a piece-wise linear function approximation to represent the speed function [5]. This approximation of speed function for a processor is built using a set of few experimentally obtained points. The block size **b** used in the experiments is 32, which is typical for cache-based workstations [3], [8].

Figure 3 shows the speedup of the LU Factorization application using the Variable Group Block distribution strategy over the application using the Group Block Distri-

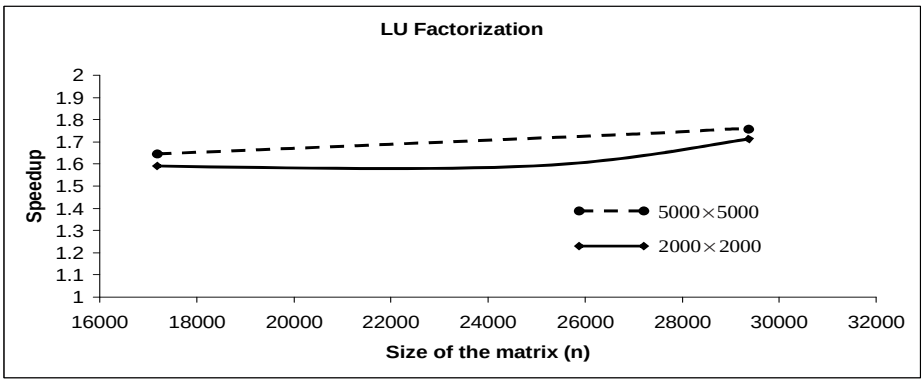


**Fig. 3.** Speedup of Variable Group Block Distribution over Group Block Distribution. For the Group Block Distribution, the single number speeds are obtained using DGEMM for a dense square matrix. For the solid lined curve, the matrix used is of size 2000×2000. For the dashed curve, the matrix used is of size 5000×5000.

bution strategy. The speedup calculated is the ratio of the execution time of the LU Factorization application using the Group Block distribution strategy over the execution time of the application using the Variable Group Block Distribution strategy.

## 4 Conclusions and future work

In this paper, we presented a static data distribution strategy called Variable Group Block distribution to optimize the execution of factorization of a dense matrix on a network of heterogeneous computers. The distribution is based on a functional performance model of computers, which integrates some of the essential features underlying applications run on general-purpose common heterogeneous networks, such as the processor heterogeneity in terms of the speeds of the processors, the memory heterogeneity in terms of the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy, and the effects of paging.

Future work would involve extension of Variable Group Block distribution strategy to optimize the execution of factorization of a dense matrix on a heterogeneous network of computers using a functional model that would incorporate communication cost parameters, namely, latency and the bandwidth of the communication links interconnecting the processors.

## References

1. Arapov, D., Kalinov, A., Lastovetsky, A., Ledovskih, I.: Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization. Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98), Lecture Notes in Computer Science, Vol. 1457, (1998) 332–343

2. Kalinov, A., Lastovetsky, A.: Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers. Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99), Lecture Notes in Computer Science, Vol. 1593, (1999) 191–200
3. Barbosa, J., Tavares, J., Padilha, A.J.: Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers. Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000) 147-159
4. Barbosa, J., Morais, C.N., Padilha, A.J.: Simulation of Data Distribution Strategies for LU Factorization on Heterogeneous Machines. Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)
5. Lastovetsky, A., Reddy, R.: Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)
6. Choi, J., Dongarra, J., Ostrouchov, L.S., Petitet, A.P., Walker, D.W., Whaley, R.C.: The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Rou-tines. Scientific Programming, Vol. 5, (1996) 173-184
7. Beaumont, O., Boudet, V., Petitet, A., Rastello, F., Robert, Y.: A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). IEEE Transactions on Computers, Vol. 50, (2001) 1052–1070
8. Blackford, L.S., Choi, J., Cleary, A., DAzevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users Guide. SIAM (1997)