Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 33 (2007) 757–779

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Data distribution for dense factorization on computers with memory heterogeneity

Alexey Lastovetsky *, Ravi Reddy

School of Computer Science and Informatics, UCD, Belfield, Dublin 4, Ireland

Received 22 November 2006; received in revised form 29 March 2007; accepted 1 June 2007
Available online 14 June 2007

Abstract

In this paper, we study the problem of optimal matrix partitioning for parallel dense factorization on heterogeneous processors. First, we outline existing algorithms solving the problem that use a constant performance model of processors, when the relative speed of each processor is represented by a positive constant. We also propose a new efficient algorithm, called the Reverse algorithm, solving the problem with the constant performance model. We extend the presented algorithms to the functional performance model, representing the speed of a processor by a continuous function of the task size. The model, in particular, takes account of memory heterogeneity and paging effects resulting in significant variations of relative speeds of the processors with the increase of the task size. We experimentally demonstrate that the functional extension of the Reverse algorithm outperforms functional extensions of traditional algorithms.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Heterogeneous systems; Scheduling and task partitioning; Load balancing and task assignment; Data distribution; Parallel algorithms; LU factorization

1. Introduction

The paper presents a static data distribution strategy for factorization of a large dense matrix on a cluster of computers with memory heterogeneity that uses a functional performance model of the computers and an algorithm of set partitioning with this model. The functional model captures different aspects of heterogeneity of the computers including the heterogeneity of the memory structure and paging effects.

A number of distribution strategies for matrix factorization in heterogeneous environments have been designed and implemented. Arapov et al. [1] propose a distribution strategy for 1D parallel Cholesky factorization. They consider the Cholesky factorization to be an irregular problem and distribute data amongst the processors of the executing parallel machine in accordance with their relative speeds. The distribution strategy divides the matrix into a number of column panels such that the width of each column panel is proportional to the speed of the processor. This strategy is developed into a more general 2D distribution strategy in [2].

* Corresponding author. Tel.: +353 1 716 2916; fax: +353 1 269 7262.
E-mail addresses: Alexey.Lastovetsky@ucd.ie (A. Lastovetsky), Manumachu.Reddy@ucd.ie (R. Reddy).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.06.001

Beaumont et al. [3] and Boulet [4] employ a dynamic programming algorithm (DP) to partition the matrix in parallel 1D LU factorization. When processor speeds are accurately known and guaranteed not to change during program execution, the dynamic programming approach provides the best possible load balancing of the processors. A static group block distribution strategy [5,6] is used in parallel 1D LU factorization to partition 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 over the processors are fixed and are determined based on speeds of the processors, which are represented by a single constant number.

All these aforementioned distribution strategies are based on a 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. The number characterizing the performance of the processor is typically its relative speed demonstrated during the execution of the code solving locally the core computational task of some given size. The single number model has proved to be accurate enough for heterogeneous distributed memory systems if partitioning of the problem results in a set of computational tasks each fitting into the main memory of the assigned processor.

But the model becomes less accurate in the following cases:

- The processors have significantly different sizes of main memory and the partitioning of the problem may result in some computational tasks not fitting into the main memory of the assigned processor. In this case, solution of the computational task of any fixed size does not guarantee accurate estimation of the relative speed of the processors. The point is that beginning from some task size, the task of the same size 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 task size exceeds the critical value.
- Even if the processors of different architectures have almost the same size of main memory, 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 task size exceeds the threshold causing the paging.

Thus, taking account of memory heterogeneity and the effects of paging significantly complicates the design of algorithms that distribute 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 be no 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 taking account of memory heterogeneity and effects of paging, a more realistic performance model of a set of heterogeneous processors is needed. A few performance models have appeared recently that take into account the memory heterogeneity. Du et al. [7] present an analytical model for evaluating the performance impact of memory hierarchies and networks on cluster computing. The model quantitatively predicts the average execution time per instruction based on the locality parameter values obtained by program memory access pattern analysis. Their study shows that the depth of the memory hierarchy is the most sensitive factor affecting the execution time for many types of workloads. The model covers only homogeneous cluster platforms. Manegold et al. [8] identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. Each level is characterized by a few parameters describing its sizes and timings. The cost functions are parameterized to accommodate various hardware characteristics appropriately. Combining the basic patterns, they describe the memory access patterns of database operations. Wang [9] studies the impact of memory access latency on load sharing problems on heterogeneous networks of workstations. In addition to CPU speed and memory capacity, the method takes memory access time into consideration and shows that memory access latency is an important consideration in the design of load sharing policies on heterogeneous networks of workstations. Rauber and Runger [10] propose a unifying computation model that models the memory access time of distributed shared-memory (DSM) systems by a memory hierarchy comprising different memories with different capacities and access times. Drozdowski and Wolniewicz [11] propose a model that considers both processor and memory heterogeneity. Their model characterizes the performance of the processor by the execution time of the task, represented by a piecewise linear function of its size, which is equivalent to representation of the speed of the processor by a unit step function of the task size. The model is targeted mainly towards carefully designed scientific codes, efficiently using memory hierarchy and running on dedicated multi-processor computer systems.

Applicability of the functional performance model of heterogeneous processors proposed in [12,13] is not limited to carefully designed applications running on dedicated systems. It can accurately describe the performance of both carefully and casually designed applications on both dedicated systems and general-purpose networks of heterogeneous computers. Under the functional model, the speed of each processor is represented by a continuous function of the task size. The speed is defined as the number of computation units performed by the processor per one time unit. The model is application centric in the sense that different applications will characterize the speed of the processor by different functions.

In this paper, an algorithm of optimal set partitioning with the functional model [13] is used as a building block in the design of efficient algorithms of matrix partitioning for parallel LU factorization on a cluster of heterogeneous processors. The data distribution algorithms, presented in this paper, use static data distribution strategy and hence do not result in redistribution of data between steps of the execution of the LU factorization application.

The functional model does not take into account the cost of communications. This factor can be ignored if the contribution of communication operations in the total execution time of the application is negligible compared to that of computations. The LU application used in this paper falls into this category. Incorporation of communication cost in the functional model and design of efficient data partitioning algorithms with such an extended model is a subject of our current research and out of scope of this paper.

The rest of the paper is organized as follows. In Section 2, we present a summary of the functional performance model and the set-partitioning algorithms with this model. In Section 3, the homogeneous LU factorization algorithm that is used for our heterogeneous modification is presented. In Section 4, two existing heterogeneous modifications of this algorithm using the constant model of heterogeneous processors are outlined and our original modification also based on the constant performance model introduced. Section 5 presents functional extensions of the three heterogeneous algorithms when distribution of columns of the matrix is based on the functional performance model of heterogeneous processors. Finally, experimental results on a local network of heterogeneous computers are presented demonstrating the efficiency of the proposed strategy.

2. Partitioning algorithms using the functional performance model

Under the functional performance model, the speed of each processor is represented by a continuous function of the task size.

The speed is defined as the number of computation units performed by the processor per one time unit. The model is application specific. In particular, this means that the computation unit can be defined differently for different applications. The important requirement is that the computation unit does not vary during the execution of the application. An arithmetical operation and the matrix update $a = a + b \times c$, where a, b, and c are $r \times r$ matrices of the fixed size r, give us examples of computation units.

The task size is understood as a set of one, two or more parameters characterizing the amount and layout of data stored and processed during the execution of the computational task (compare with the notion of problem size as the number of basic computations in the best sequential algorithm to solve the problem on a single processor [14]). The number and semantics of the task size parameters are problem or even application specific. It is assumed that the amount of stored data will increase with the increase of any of the task size parameters.

For example, the size of the task of multiplication of two dense rectangular $n \times k$ and $k \times m$ matrices can be represented by three parameters, n, k, and m. The total number of matrix elements to store and process is $(n \times k + k \times m + n \times m)$. The total number of arithmetical operations needed to solve this task is

$(2 \times k - 1) \times n \times m$. If k is large enough, the number can be approximated by $2 \times k \times n \times m$. Alternatively, a combined computation unit, which is made up of one addition and one multiplication, can be used to express this volume of computation. In this case, the total number of computation units will be approximately equal to $k \times n \times m$. Therefore, the speed of the processor demonstrated by the application when solving the task of size (n,k,m) can be calculated as $k \times n \times m$ (or $2 \times k \times n \times m$) divided by the execution time of the application. This gives us a function, $f: \mathbf{N}^3 \to \mathbf{R}_+$, mapping task sizes to speeds of the processor. The functional performance model of the processor is obtained by continuous extension of function $f: \mathbf{N}^3 \to \mathbf{R}_+$ to function g: $\mathbf{R}_+^3 \to \mathbf{R}_+$ ($f(n,k,m) = g(n,k,m)$ for any (n,k,m) from $\mathbf{N}^3$).

Thus, under the proposed functional model, the speed of the processor is represented by a continuous function of the task size. Moreover, some further assumptions can be made about the shape of the function. Namely, it can be realistically assumed that along each of the task size variables, either the function is monotonically decreasing, or there exists point x such that

- On the interval $[0, x]$, the function is
  - Monotonically increasing.
  - Concave, and
  - any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.
- On the interval $[x, \infty)$, the function is monotonically decreasing.

The results of Kitchen et al. [15] justify these assumptions. They present the results of benchmark experiments carried out on Xeon, Opteron, Itanium2 and Power5-based clusters using the distributed memory version of the EuroBen benchmark suite [16]. The distributed-memory version of the EuroBen benchmark suite contains 13 diverse programs. They compare and contrast the observed performance as a function of the task size.

Lastovetsky and Reddy [13] study the problem of optimal partitioning of an n-element set over p heterogeneous processors with the functional model and design an algorithm of its solution of the complexity $O(p \times \log_2 n)$. The low complexity is mainly due to the assumption of bounded heterogeneity of the processors. The assumption will be inaccurate if the speed of some processors becomes too slow for large n effectively approaching 0. One approach to this problem is to use a relaxed functional model where the speed of processor is represented by a continuous function until some given size of the task and by zero for all sizes greater than this one. Data partitioning algorithms with that model are presented in [17]. The other approach is to use algorithms not sensitive to the shape of performance functions such as the algorithm of complexity $O(p^2 \times \log_2 n)$ presented in [12]. The special case of heterogeneous processors whose performance is characterized by positive constants is studied in [3,4].

3. LU Factorization on homogeneous multi-processors

Before we present our distribution strategy, we describe the LU Factorization algorithm of a dense $(n \times b) \times (n \times b)$ matrix A, one step of which is shown in Fig. 1, where n is the number of blocks of size $b \times b$, optimal values of b depending on the memory hierarchy and on the communication-to-computation ratio of the target computer [18,19].

The LU factorization applies a sequence of Gaussian eliminations to form $A = P \times L \times U$, where A, L, and U are dense $(n \times b) \times (n \times b)$ matrices. P is a permutation matrix which is stored in a vector of size $n \times b$, L is unit lower triangular (lower triangular with 1's on the main diagonal), and U is upper triangular.

At the kth step of the computation ($k = 1, 2, \ldots$), it is assumed that the $m \times m$ submatrix of $A^{(k)}$ ($m = ((n - (k - 1)) \times b)$ is to be partitioned as follows:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = P \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = P \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}$$

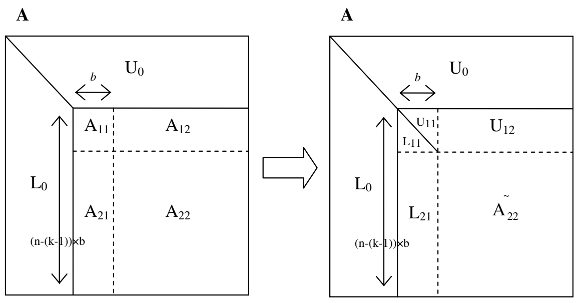

Fig. 1. One step of the LU factorization algorithm of a dense matrix A of size $(n \times b) \times (n \times b)$.

where the block A_{11} is $b \times b$, A_{12} is $b \times (m-b)$, A_{21} is $(m-b) \times b$, and A_{22} is $(m-b) \times (m-b)$. L_{11} is unit lower triangular matrix, and U_{11} is an upper triangular matrix.

At first, a sequence of Gaussian eliminations is performed on the first $m \times b$ panel of $A^{(k)}$ (i.e., A_{11} and A_{21}). Once this is completed, the matrices L_{11}, L_{21}, and U_{11} are known and we can rearrange the block equations

$$U_{12} \leftarrow (L_{11})^{-1} A_{12},$$
$$\widetilde{A}_{22} \leftarrow A_{22} L_{21} U_{12} = L_{22} U_{22}.$$

The LU factorization can be done by recursively applying the steps outlined above to the $(m-b) \times (m-b)$ matrix $\widetilde{A}_{22}$. Fig. 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. In the figure, the regions L_0, U_0, L_{11}, U_{11}, L_{21}, and U_{12} represent data for which the corresponding computations are completed. Later row interchanges will be applied to L_0 and L_{21}.

Now we present a parallel algorithm that computes the above steps on a one-dimensional arrangement of p homogeneous processors. The algorithm can be summarized as follows:

1. A CYCLIC(b) distribution of columns is used to distribute the matrix A over a one-dimensional arrangement of p homogeneous processors as shown in Fig. 2. The cyclic distribution assigns columns of blocks with numbers $1,2,\ldots,n$ to processors $1,2,\ldots,p,1,2,\ldots,p,1,2,\ldots$, respectively, for a p-processor linear array ($n \gg p$), until all n columns of blocks are assigned.
2. The algorithm consists of n steps. At each step ($k = 1,2,\ldots$),

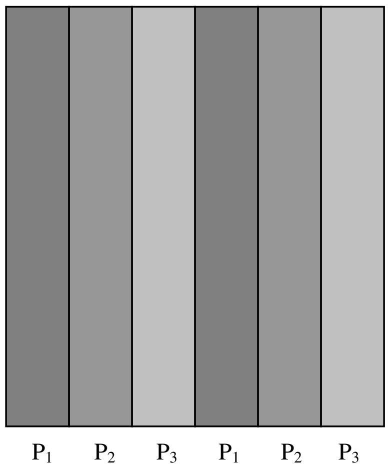

Fig. 2. Column-oriented CYCLIC distribution of six column blocks on a one-dimensional array of three homogeneous processors.

- The processor owning the pivot column block of the size $((n-(k-1))\times b)\times b$ (i.e., A_{11} and A_{21}) factors it;
- All processors apply row interchanges to the left and the right of the current column block k:
- The processor owning L_{11} broadcasts it to the rest of the processors, which convert the row panel A_{12} to U_{12}.
- The processor owning the column panel L_{21} broadcasts it to the rest of the processors.
- All the processors update their local portions of the matrix, A_{22}, in parallel.The implementation of the algorithm, which is used in the paper, is based on the ScaLAPACK [19] routine, **PDGETRF**, and consists of the following steps:

1. **PDGETF2**: Apply the LU factorization to the pivot column panel of size $((n-(k-1))\times b)\times b$ (i.e.,A_{11} and A_{21}). It should be noted here that only the routine **PDSWAP** employs all the processes involved in the parallel execution. The rest of the routines are performed locally at the process owning the pivot column panel.
   - [Repeat b times $(i = 1,\dots,b)$]
     - **PDAMAX**: find the (absolute) maximum element of the ith column and its location.
     - **PDSWAP**: interchange the ith row with the row that holds the maximum.
     - **PDSCAL**: scale the ith column of the matrix.
     - **PDGER**: update the trailing submatrix.
   - The process owning the pivot column panel broadcasts the same pivot information to all the other processes.
2. **PDLASWP**: All processes apply row interchanges to the left and the right of the current panel.
3. **PDTRSM**: L_{11} is broadcast to the other processes, which convert the row panel A_{12} to U_{12}.
4. **PDGEMM**: The column panel L_{21} is broadcast to all the other processes. Then, all processes update their local portions of the matrix, A_{22}.

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 its update. 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 their balanced load.

4. LU factorization on heterogeneous platforms with a constant performance model of processors

Heterogeneous parallel algorithms of LU factorization on heterogeneous platforms are obtained by modification of the homogeneous algorithm presented in Section 3. The modification is in the distribution of column panels of matrix A over the linear array of processors. As the processors are heterogeneous having different speeds, the optimal distribution that aims at balancing the updates at all steps of the parallel LU factorization will not be fully cyclic. So, the problem of LU factorization of a matrix on a heterogeneous platform is reduced to the problem of distribution of column panels of the matrix over heterogeneous processors of the platform.

Traditionally, the distribution problem is formulated as follows: Given a dense $(n\times b)\times(n\times b)$ matrix A, how can we assign n columns of size $n\times b$ of the matrix A to p $(n \gg p)$ heterogeneous processors $P_1, P_2, \dots P_p$ of relative speeds $S = \{s_1, s_2, \dots, s_p\}$, $\sum_{i=1}^{p} s_i = 1$, so that the workload at each step of the parallel LU factorization is best balanced? The relative speed s_i of processor P_i is obtained by normalization of its (absolute) speed a_i, understood as the number of column panels updated by the processor per one time unit, $s_i = a_i / \sum_{i=1}^{p} a_i$. To explain what "the number of column panels updated by the processor P_i" means, let us first consider the first step $(k = 1)$ of the LU factorization. Remember that at step k $(k = 1,2,\dots)$ of the LU factorization, column panels of size $(n-(k-1))\times b$ are updated. Assuming the partitioning $\sum_{i=1}^{p} n_i^{(1)} = n$ for the first step, where $n_i^{(1)}$ denotes the number of column panels allocated to processor P_i, the number of column panels of size $n\times b$ updated by the processor P_i will be $n_i^{(1)}$. Next, assuming the partitioning $\sum_{i=1}^{p} n_i^{(2)} = n-1$ for the second step $(k = 2)$ of the LU factorization, where $n_i^{(2)}$ denotes the number of column panels allocated to processor P_i, the number of column panels of size $(n-1)\times b$ updated by the processor P_i is $n_i^{(2)}$, and so on. While a_i will increase with each next step of the LU factorization (because the height of updated column panels will decrease as the LU factorization progresses, resulting in a larger number of column panels updated by the processor per time unit), the relative speeds s_i are assumed to be constant. The optimal solution sought is the one that minimizes $\max_i n_i^{(k)}/s_i$ for each step of the LU factorization ($\sum_{i=1}^{p} n_i^{(k)} = n^{(k)}$), where $n^{(k)}$ is the total number of column panels updated at the step k and $n_i^{(k)}$ denotes the number of column panels allocated to processor P_i.

The motivation behind that formulation is the following. Strictly speaking, the optimal solution should minimize the total execution time of the LU factorization, which is given by $\sum_{k=1}^{n}\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)}$, where $a_i^{(k)}$ is the speed of processor P_i at step k of the LU factorization and $n_i^{(k)}$ is the number of column panels updated by processor P_i at this step. However, if a solution minimizes $\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)}$ for each k, it will also minimize $\sum_{k=1}^{n}\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)}$. Because $\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)} = \max_{i=1}^{p} n_i^{(k)}/(s_i \times \sum_{i=1}^{p} a_i^{(k)}) = (1/\sum_{i=1}^{p} a_i^{(k)}) \times \max_{i=1}^{p} n_i^{(k)}/s_i$, then for any given k the problem of minimization of $\sum_{k=1}^{n}\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)}$ will be equivalent to the problem of minimization of $\max_{i=1}^{p} n_i^{(k)}/s_i$. Therefore, if we are lucky and there exists an allocation that minimizes $\max_{i=1}^{p} n_i^{(k)}/s_i$ for each step k of the LU factorization, then the allocation will be globally optimal, minimizing $\sum_{k=1}^{n}\max_{i=1}^{p} n_i^{(k)}/a_i^{(k)}$. Fortunately, such an allocation does exist [3,4].

Now we briefly outline two existing approaches to solution of the above distribution problem, which are the Group Block (GB) distribution algorithm [5] and the Dynamic Programming (DP) distribution algorithm [3,4].

The GB algorithm. This algorithm partitions the matrix into groups (or *generalized blocks* in terms of [2]), all of which have the same number of column panels. The number of column panels per group (the size of the group) and the distribution of the column panels within the group over the processors are fixed and determined based on relative speeds of the processors. The relative speeds are obtained by running the DGEMM routine that locally updates some particular dense rectangular matrix. The inputs to the algorithm are p, the number of heterogeneous processors in the one-dimensional arrangement, b, the block size, n, the size of the matrix in number of blocks of size $b \times b$ or the number of column panels, and $S = \{s_1, s_2, \ldots, s_p\}\left(\sum_{i=1}^{p} s_i = 1\right)$, the relative speeds of the processors. The outputs are g, the size of the group, and d, an integer array of size p, the ith element of which contains the number of column panels in the group assigned to processor i. The algorithm can be summarized as follows:

(1) The size of the group g is calculated as $\lfloor 1/\min(s_i) \rfloor$ $(1 \leqslant i \leqslant p)$. If $g/p < 2$, then $g = \lfloor 2/\min(s_i) \rfloor$. This condition is imposed to ensure there is sufficient number of blocks in the group.
(2) The group is partitioned so that the number of column panels d_i assigned to processor i in the group will minimize $\max_i \frac{d_i}{s_i}$ (see [5] for a simple algorithm performing this partitioning).
(3) In the group, processors are reordered to start from the slowest processors to the fastest processors for load balance purposes.

The complexity of this algorithm is $O(p \times \log_2 p)$. At the same time, the algorithm does not guarantee that the returned solution will be optimal.

The DP algorithm. Dynamic programming is used to distribute column panels of the matrix over the processors. The relative speeds of the processors are obtained by running the DGEMM routine that locally updates some particular dense rectangular matrix. The inputs to the algorithm are p, the number of heterogeneous processors in the one-dimensional arrangement, b, the block size, n, the size of the matrix in number of blocks of size $b \times b$ or the number of column panels, and $S = \{s_1, s_2, \ldots, s_p\}\left(\sum_{i=1}^{p} s_i = 1\right)$, the relative speeds of the processors. The outputs are c, an integer array of size p, the ith element of which contains the number of column panels assigned to processor i, and d, an integer array of size n, the ith element of which contains the processor to which the column panel i is assigned. The algorithm can be summarized as follows:

```
$(c_1, \ldots, c_p) = (0, \ldots, 0)$;
$(d_1, \ldots, d_n) = (0, \ldots, 0)$;
```

```
for($k = 1$; $k \leqslant n$; $k = k + 1$) {
    $\text{Cost}_{\min} = \infty$;
    for($i = 1; i <= p; i = i + 1$) {
            Cost=$(c_i + 1)/s_i$;
            if (Cost < $\text{Cost}_{\min}$) {$\text{Cost}_{\min}$ = Cost; $j = i$;}
    }
    $d_{n-k+1} = j$;
    $c_j = c_j + 1$;
}
```

The complexity of the DP algorithm is O($p \times n$). The algorithm returns the optimal allocation of the column panels to the heterogeneous processors [3,4]. The fact that the DP algorithm always returns the optimal solution is not trivial. Indeed, at each iteration of the algorithm the column panel k is allocated to one of the processors, namely, to a processor, minimizing the cost of the allocation. At the same time, there may be several processors with the same, minimal, cost of allocation. The algorithm randomly selects one of them. It is not obvious that allocation of the column panel to any of these processors will result in a globally optimal allocation. But, fortunately, for this particular distribution problem this is proved to be true.

In this paper, we propose another algorithm solving this distribution problem, a Reverse distribution algorithm. Like the DP algorithm, the Reverse algorithm always returns the optimal allocation. The complexity of the Reverse algorithm,O($p \times n \times \log_2 p$), is a bit worse than that of the DP algorithm, but the algorithm has one important advantage. It is better suitable as a basis for extensions dealing with the functional performance model of heterogeneous processors.

The Reverse algorithm. This algorithm generates the optimal distribution $(n_1^{(k)}, \ldots, n_p^{(k)})$ of $n \times b$ column panels of the dense $(n \times b) \times (n \times b)$ matrix over p heterogeneous processors for each step k of the parallel LU factorization $(\sum_{i=1}^{p} n_i^{(k)} = n - k + 1, k = 1, \ldots, n)$ and then allocates the column panels to the processors by comparing these distributions. In other words, the algorithm extracts the optimal allocation of the column panels from a sequence of optimal distributions of the panels for successive steps of the parallel LU factorization. The inputs to the algorithm are p, the number of heterogeneous processors in the one-dimensional arrangement, b, the block size, n, the size of the matrix in number of blocks of size $b \times b$ or the number of column panels, and $S = \{s_1, s_2, \ldots, s_p\} \left(\sum_{i=1}^{p} s_i = 1\right)$, the relative speeds of the processors. The output is d, an integer array of size n, the ith element of which contains the processor to which the column panel i is assigned. The algorithm can be summarized as follows:

```
$(d_1, \ldots, d_n) = (0, \ldots, 0)$;
$w = 0$;
$(n_1, \ldots, n_p) = \text{HSP}(p, n, S)$;
for ($k = 1$; $k < n$; $k = k + 1$) {
    $(n'_1, \ldots, n'_p) = \text{HSP}(p, n - k, S)$;
    if ($w == 0$)
    then if $((\exists! j \in [1, p])(n_j == n'_j + 1) \wedge (\forall i \neq j)(n_i == n'_i))$
        then {$d_k = j$; $(n_1, \ldots, n_p) = (n'_1, \ldots, n'_p)$;}
        else $w = 1$;
    else if $((\exists i \in [1])(n_i < n'_i))$
        then $w = w + 1$;
        else {
            for ($i = 1$; $i \leqslant p$; $i = i + 1$)
                for ($\Delta = n_i - n'_i$; $\Delta \neq 0$; $\Delta = \Delta - 1$, $w = w - 1$)
                    $d_{k-w} = i$;
                $(n_1, \ldots, n_p) = (n'_1, \ldots, n'_p)$;
                $w = 0$;
            }
    }
```

```
If ((∃$i \in [1,p]$)($n_i == 1$))
then $d_n = i$;
```

Here, HSP(p, n, S) returns the optimal distribution of n column panels over p heterogeneous processors of the relative speeds $S = \{s_1, s_2, \ldots, s_p\}$ by applying the algorithm for optimal distribution of independent chunks of computations from [3,4] (HSP stands for Heterogeneous Set Partitioning). Thus, first we find the optimal distributions of column panels for the first and second steps of the parallel LU factorization. If the distributions differ only for one processor, then we assign the first column panel to this processor. The reason is that this assignment guarantees a transfer from the best workload balance at the first step of the LU factorization to the best workload balance at its second step. If the distributions differ for more than one processor, we postpone allocation of the first column panel and find the optimal distribution for the third step of the LU factorization and compare it with the distribution for the first step. If the number of panel columns distributed to each processor for the third step does not exceed that for the first step, we allocate the first and second column panels so that the distribution for each next step is obtained from the distribution for the immediate previous step by addition of one more column panel to one of the processors. If not, we delay allocation of the first two column panels and find the optimal distribution for the fourth step and so on.

In Table 1, we demonstrate the algorithm for $n = 10$. The first column represents the step k of the algorithm. The second column shows the distributions obtained during each step by HSP. The entry "Allocation made" denotes the rank of the processor to which the column panel k is assigned. At steps $k = 4$ and $k = 5$, the algorithm does not make any assignments. At $k = 6$, processor P_1 is allocated column panels (4, 5) and processor P_2 is allocated column panel 6. The output d in this case would be $(P_1 P_1 P_1 P_1 P_1 P_3 P_2 P_1 P_2 P_3)$.

Proposition 1. *The Reverse algorithm returns the optimal allocation.*

Proof of Proposition 1. *If the algorithm assigns the column panel k at each iteration of the algorithm, then the resulting allocation will be optimal by design. Indeed, in this case the distribution of column panels over the processors will be produced by the HSP and hence optimal for each step of the LU factorization.*

Consider the situation when the algorithm assigns a group of w $(w > 1)$ column panels beginning from the column panel k. In that case, the algorithm first produces a sequence of $(w + 1)$ distributions $(n_1^{(k)}, \ldots, n_p^{(k)})$, $(n_1^{(k+1)}, \ldots, n_p^{(k+1)}), \ldots, (n_1^{(k+w)}, \ldots, n_p^{(k+w)})$ such that

- *the distributions are optimal for steps k, $k + 1, \ldots, k + w$ of the LU factorization respectively, and*
- *$(n_1^{(k)}, \ldots, n_p^{(k)}) > (n_1^{(k+i)}, \ldots, n_p^{(k+i)})$ is only true for $i = w$ (by definition, $(a_1, \ldots, a_p) > (b_1, \ldots, b_p)$ if and only if $(\forall i)(a_i \geqslant b_i) \wedge (\exists i)(a_i > b_i)$).*

Lemma 1. *Let $(n_1, \ldots, n_p)$ and $(n'_1, \ldots, n'_p)$ be optimal distributions such that $n = \sum_{i=1}^{p} n_i > \sum_{i=1}^{p} n'_i = n'$, $(\exists i)(n_i < n'_i)$ and $(\forall j)(\max_{i=1}^{p} n_i/s_i \leqslant (n_j + 1)/s_j)$. Then, $\max_{i=1}^{p} n_i/s_i = \max_{i=1}^{p} n'_i/s_i$.*

Table 1
Reverse algorithm with three processors P_1, P_2, P_3

Step of the algorithm (k)	Distributions at step k			Allocation made
	P_1	P_2	P_3	
	6	2	2	
1	5	2	2	P_1
2	4	2	2	P_1
3	3	2	2	P_1
4	1	3	2	No allocation
5	1	3	1	No allocation
6	1	2	1	P_1, P_1, P_3
7	1	1	1	P_2
8	0	1	1	P_1
9	0	0	1	P_2
10				P_3

Proof of Lemma 1. *As* $n > n'$ *and* $(n_1, \ldots, n_p)$ *and* $(n'_1, \ldots, n'_p)$ *are both optimal distributions, then* $\max_{i=1}^{p} n_i/s_i \geqslant \max_{i=1}^{p} n'_i/s_i$. *On the other hand, there exists* $j \in [1,p]$ *such that* $n_j < n'_j$, *which implies* $n_j + 1 \leqslant n'_j$. *Therefore,* $\max_{i=1}^{p} n'_i/s_i \geqslant n'_j/s_j \geqslant (n_j + 1)/s_j$. *As we assumed that* $(\forall j)(\max_{i=1}^{p} n_i/s_i \leqslant (n_j + 1)/s_j)$, *then* $\max_{i=1}^{p} n_i/s_i \leqslant (n_j + 1)/s_j \leqslant n'_j/s_j \leqslant \max_{i=1}^{p} n'_i/s_i$. *Thus, from* $\max_{i=1}^{p} n_i/s_i \geqslant \max_{i=1}^{p} n'_i/s_i$ *and* $\max_{i=1}^{p} n_i/s_i \leqslant \max_{i=1}^{p} n'_i/s_i$ *we conclude that* $\max_{i=1}^{p} n_i/s_i = \max_{i=1}^{p} n'_i/s_i$. □

We can apply Lemma 1 to the pair $(n_1^{(k)}, \ldots, n_p^{(k)})$ and $(n_1^{(k+l)}, \ldots, n_p^{(k+l)})$ for any $l \in [1, w-1]$. Indeed, $\sum_{i=1}^{p} n_i^{(k)} > \sum_{i=1}^{p} n_i^{(k+l)}$ and $(\exists i)(n_i^{(k)} < n_i^{(k+l)})$. Finally, the HSP guarantees that $(\forall j)(\max_{i=1}^{p} n_i^{(k)}/s_i \leqslant (n_j^{(k)} + 1)s_j)$ (see [3,4]). Therefore, $\max_{i=1}^{p} n_i^{(k)}/s_i = \max_{i=1}^{p} n_i^{(k+1)}/s_i = \ldots = \max_{i=1}^{p} n_i^{(k+w-1)}/s_i$. In particular, this means that for any $(m_1, \ldots, m_p)$ such that $\min_{j=k}^{k+w-1} n_i^{(j)} \leqslant m_i \leqslant \max_{j=k}^{k+w-1} n_i^{(j)}$ $(i = 1, \ldots, p)$, we will have $\max_{i=1}^{p} m_i/s_i = \max_{i=1}^{p} n_i^{(k)}/s_i$. The allocations made in the end by the Reverse algorithm for the column panels $k, k+1, \ldots, k+w-1$ result in a new sequence of distributions for steps $k, k+1, \ldots, k+w-1$ of the LU factorization such that each next distribution differs from the previous one for exactly one processor. Each distribution $(m_1, \ldots, m_p)$ in this new sequence satisfies the inequality $\min_{j=k}^{k+w-1} n_i^{(j)} \leqslant m_i \leqslant \max_{j=k}^{k+w-1} n_i^{(j)}$ $(i = 1, \ldots, p)$. Therefore, all they will have the same cost $\max_{i=1}^{p} n_i^{(k)}/s_i$, which is the cost of the optimal distribution for these steps of the LU factorization found by the HSP. Hence, each distribution in this sequence will be optimal for the corresponding step of the LU factorization. □

Proposition 2. *The complexity of the Reverse algorithm is* $O(p \times n \times \log_2 p)$.

Proof. At each iteration of this algorithm, we apply the HSP, which is of complexity $O(p \times \log_2 p)$ [3,4]. Testing the condition $(\exists! j \in [1])(n_j == n'_j + 1) \wedge (\forall i \neq j)(n_i == n'_i)$ is of complexity $O(p)$. Testing the condition $(\exists i \in [1,p])(n_i < n'_i)$ is also of complexity $O(p)$. Finally, the total number of iterations of the inner loop of the nest of loops.

```
for ($i = 1$; $i \leqslant p$; $i = i + 1$)
    for ($\Delta = n_i - n'_i$; $\Delta \neq 0$; $\Delta = \Delta - 1$, $w = w - 1$)
        $d_{k-w} = i$;
```

during the execution of the algorithm cannot exceed the total number of allocations of column panels, n. Thus, the overall complexity of the algorithm is upper bounded by $n \times O(p \times \log_2 p) + n \times O(p) + n \times O(p) + p \times n \times O(1) = O(p \times n \times \log_2 p)$. □

5. LU Factorization on heterogeneous platforms with the functional performance model of processors

The problem of distribution of a dense square matrix over heterogeneous processors for the parallel LU factorization, as it is formulated in Section 4, uses the constant performance model of processors that does not address the memory heterogeneity and paging effects. The functional performance model [12,13] addresses the issues. Under this model, the speed of each computer is represented by a continuous and relatively smooth function of the task size. This model is application centric in the sense that, generally speaking, different applications will characterize the speed of the processor by different functions.

In this section, we formulate the problem of optimal distribution of column panels of a dense square matrix for its efficient LU factorization over a one-dimensional arrangement of heterogeneous processors with their functional performance model. Then we describe how the distribution algorithms presented in Section 4 can be modified for solution of the functional distribution problem.

5.1. Problem formulation

The problem of distributing a large dense square matrix A for its parallel LU factorization over a one-dimensional arrangement of heterogeneous processors using their functional performance model can be formulated as follows:

- Given
  - A dense $n \times n$ matrix A, and
  - p $(n > p)$ heterogeneous processors P_1, $P_2, \ldots, P_p$ of respective speeds $S = \{s_1(x,y),\ s_2(x,y), \ldots, s_p(x,y)\}$, where $s_i(x,y)$ is the speed of the update of an $x \times y$ matrix by the processor i, measured in arithmetical operations per time unit and represented by a continuous function $R_+ \times R_+ \to R_+$,
- Assign the columns of the matrix A to the p processors so that the assignment minimizes $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{V(n-k, n_i^{(k)})}{s_i(n-k, n_i^{(k)})}$, where $V(x,y)$ is the number of operations needed to update a $x \times y$ matrix and $n_i^{(k)}$ is the number of columns updated by the processor P_i at step k of the parallel LU factorization.

This formulation is motivated by the n-step parallel LU factorization algorithm. One element of matrix A may represent a single number with computations measured in arithmetical operations. Alternatively, it can represent a square matrix block of a given fixed size $b \times b$. In this case, computations are measured in $b \times b$ matrix operations. The formulation assumes that the computation cost is determined by the update of the trailing matrix and fully ignores the communication cost. Therefore, the execution time of the kth step of the LU factorization is estimated by $\max_{i=1}^{p} \frac{V(n-k, n_i^{(k)})}{s_i(n-k, n_i^{(k)})}$, and the optimal solution has to minimize the overall execution time, $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{V(n-k, n_i^{(k)})}{s_i(n-k, n_i^{(k)})}$.

Unlike the constant optimization problem considered in Section 4, the functional optimization problem cannot be reduced to the problem of minimization of the execution time of all n individual steps of the LU factorization. Correspondingly, this functional matrix-partitioning problem cannot be reduced to a problem of partitioning a well-ordered set. The reason is that in the functional case there may be no globally optimal allocation of columns minimizing the execution time of all individual steps of the LU factorization. This complication is introduced by the use of the functional performance model of heterogeneous processors instead of the constant one. A simple example supporting this statement is designed as follows.

Consider an example with three processors $\{P_1, P_2, P_3\}$ distributing nine columns. Table 2 shows the functional performance models of the processors $S = \{s_1(x,y), s_2(x,y), s_3(x,y)\}$ where $s_i(x,y)$ is the speed of the update of a $x \times y$ matrix by the processor P_i. Table 3 shows the distribution of these nine columns demonstrating that there may be no globally optimal allocation of columns that minimizes the execution time of all steps of the LU factorization. The first column of the table represents the step k of the parallel LU factorization. The second column shows the global allocation of columns minimizing the total execution time of LU factorization. The third column shows the execution time of the step k of the LU factorization resulting from this allocation. The execution time $t_i^{(k)}$ for a processor i needed to update a matrix of size$(9-k) \times n_i^{(k)}$ is calculated as $\frac{V(9-k, n_i^{(k)})}{s_i(9-k, n_i^{(k)})} = \frac{(9-k) \times n_i^{(k)}}{s_i(9-k, n_i^{(k)})}$ where $n_i^{(k)}$ denotes the number of columns updated by the processor P_i (formula for the volume of computations explained below). The fourth column shows the distribution of columns, which results in the minimal execution time to solve the task size $(9-k, 9-k)$ at step k of the LU factorization. This distribution is determined by considering all possible mappings and choosing the one, which results in minimal execution time. The fifth column shows these minimal execution times for the task size $(9-k, 9-k)$. For example, consider the step $k = 2$, the local optimal distribution resulting in the minimal execution time for

Table 2
Functional model of three processors P_1, P_2, P_3

Task sizes (x,y)	$s_1(x,y)$	$s_2(x,y)$	$s_3(x,y)$
(1,1), (1,2), (1,3), (1,4),(1,5), (1,6), (1,7), (1,8)	6, 6, 6, 6, 6, 4, 4, 4	18, 18, 18, 18, 18, 18, 18, 2	**18**, 18, 18, 18, 18, 18, 18, 2
(2,1), (2,2), (2,3), (2,4), (2,5), (2,6), (2,7), (2,8)	6, 6, 6, 6, 5, 4, 3, 3	18, 18, 18, 18, 9, 8, 8, 2	18, **18**, 18, 18, 15, 12, 8, 2
(3,1), (3,2), (3,3), (3,4), (3,5), (3,6), (3,7), (3,8)	6, 6, 6, 5, 4, 3, 3, 3	**18**, 18, 18, 9, 8, 8, 6, 2	18, **18**, 18, 12, 8, 8, 8, 2
(4,1), (4,2), (4,3), (4,4), (4,5), (4,6), (4,7), (4,8)	6, 6, 5, 4, 3, 3, 3, 3	18, **18**, 9, 9, 8, 6, 5, 2	18, **18**, 12, 9, 8, 6, 6, 2
(5,1), (5,2), (5,3), (5,4), (5,5), (5,6), (5,7), (5,8)	**6**, 5, 4, 3, 3, 3, 2, 2	18, **9**, 8, 8, 6, 5, 3, 1	18, **15**, 8, 8, 6, 5, 5, 1
(6,1), (6,2), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8)	**4**, 4, 3, 3, 3, 2, 1, 1	18, **8**, 8, 6, 5, 3, 2, 1	18, 12, **8**, 6, 5, 3, 3, 1
(7,1), (7,2), (7,3), (7,4), (7,5), (7,6), (7,7), (7,8)	4, **3**, 3, 3, 2, 1, 1, 1	18, 8, **8**, 6, 5, 3, 2, 1	18, **8**, 8, 6, 5, 3, 2, 1
(8,1), (8,2), (8,3), (8,4), (8,5), (8,6), (8,7), (8,8)	4, 3, 3, **3**, 2, 1, 1, 1	2, 2, **2**, 2, 1, 1, 1, 1	2,**2**, 2, 2, 1, 1, 1, 1

Table 3
Distribution of nine column panels over three processors P_1, P_2, P_3

Step of LU factorization (k)	Global allocation of columns minimizing the overall execution time	Execution time of LU at step k	Local optimal distribution $\{n_1^{(k)}, n_2^{(k)}, n_3^{(k)}\}$ for task size $(9-k, 9-k)$	Minimum possible execution time for task size $(9-k, 9-k)$
1	$P_1\ P_1\ P_1\ P_1\ P_2\ P_3\ P_2\ P_3$	8	{4,2,2}	8
2	$\boldsymbol{P_1P_1P_1P_2P_3P_2P_3}$	**7**	{2,3,2}	**14/3**
3	$\boldsymbol{P_1P_1P_2P_3P_2P_3}$	**3**	{1,2,3}	**3/2**
4	$P_1\ P_2\ P_3\ P_2\ P_3$	10/9	{1,2,2}	10/9
5	$P_2\ P_3\ P_2\ P_3$	4/9	{0,2,2}	4/9
6	$P_3\ P_2\ P_3$	1/3	{0,1,2}	1/3
7	$P_2\ P_3$	1/9	{0,1,1}	1/9
8	P_3	1/18	{0,0,1}	1/18
Total execution time of LU factorization		**20**		

the task size $\{7,7\}$ is $\{P_1P_1P_2P_2P_2P_3P_3\}$, the speeds given by the speed functions S shown in Table 2 are $\{3,8,8\}$. So the number of columns assigned to processors $\{P_1,P_2,P_3\}$ are $\{2,3,2\}$, respectively. The execution times are $\left\{\frac{7\times2}{3},\frac{7\times3}{8},\frac{7\times2}{8}\right\}=\left\{\frac{14}{3},\frac{21}{8},\frac{14}{8}\right\}$. The execution time to solve the task size {7, 7} is the maximum of these execution times, $\frac{14}{3}$.

Consider again the step $k = 2$ shown in bold in the Table 3. It can be seen that the global optimal allocation shown in second column does not result in the minimal execution time for the task size at this step, which is $\{7,7\}$. The execution time of the LU factorization at this step based on the global optimal allocation is 7 whereas the minimal execution time given by the local optimal distribution for the task size $\{7,7\}$ at this step is 14/3.

5.2. Functional Reverse algorithm

Now the Reverse distribution algorithm is extended to obtain a Functional Reverse (FR) distribution algorithm that uses a functional performance model where the absolute speed of each processor is represented by a function of two variables representing the task size. The task used for calculation of the absolute speed is the operation $\widetilde{A}_{22} \leftarrow A_{22}L_{21}U_{12}$, which is the update of the rectangular $x\times y$ matrix A_{22} where L_{21} is a column of the size x and U_{12} is a row of the size y (remember that generally speaking the matrix elements represent $b\times b$ matrix blocks). This computation task represents the lion's share of computations performed by the processor at each step of the parallel algorithm. The absolute speed is calculated by dividing the total number of computation units performed during the execution of the task by the execution time. The total number of computation units (namely, multiplications of two $b\times b$ matrices) performed during the execution of the task is given by $x\times y$. Therefore, the speed of the processor exposed by the application when solving the task of size (x,y) can be calculated as $x\times y$ divided by the execution time of the application. Fig. 3 depicts this function for one of the computers, 'hcl11', used in experiments. Fig. 4 shows the relative speed of two computers, 'hcl09' and 'hcl02', used in experiments calculated as the ratio of their absolute speeds. One can see that the relative speed varies significantly depending on the value of variables x and y.

The main idea behind the Functional Reverse algorithm is that all allocations of columns are made using the functional performance model giving accurate estimation of the speed of the processors at each step of the LU factorization depending on the number of columns of the trailing matrix updated by each processor at this step.

The FR algorithm. This algorithm extends the Reverse algorithm presented earlier by using the functional model of heterogeneous processors. The inputs to the algorithm are

- p, the number of heterogeneous processors in the one-dimensional arrangement,
- n, the size of the matrix,
- $S=\{s_1(x,y),s_2(x,y),\ldots,s_p(x,y)\}$, the speed functions of the processors, and

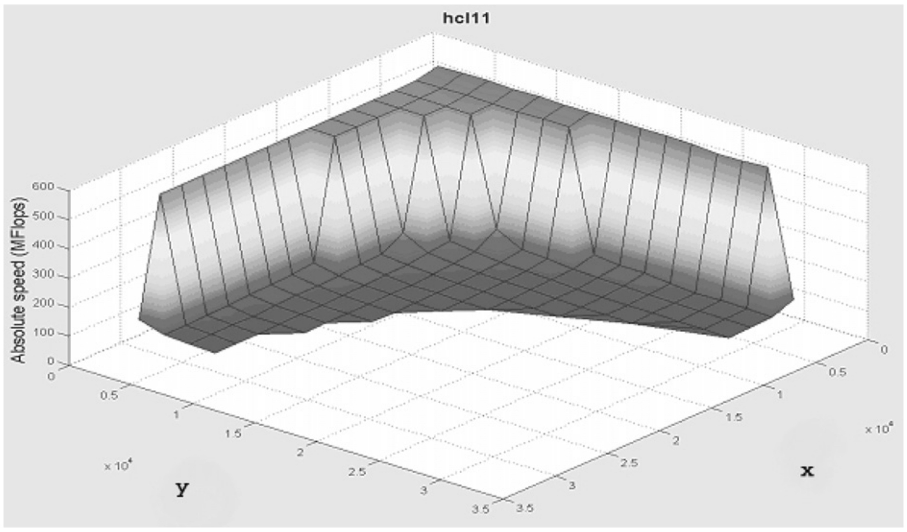

Fig. 3. The absolute speed of a computer ‘hcl11’ as a function of the size of the computational task of updating a dense $x \times y$ matrix using level-3 BLAS routine DGEMM.

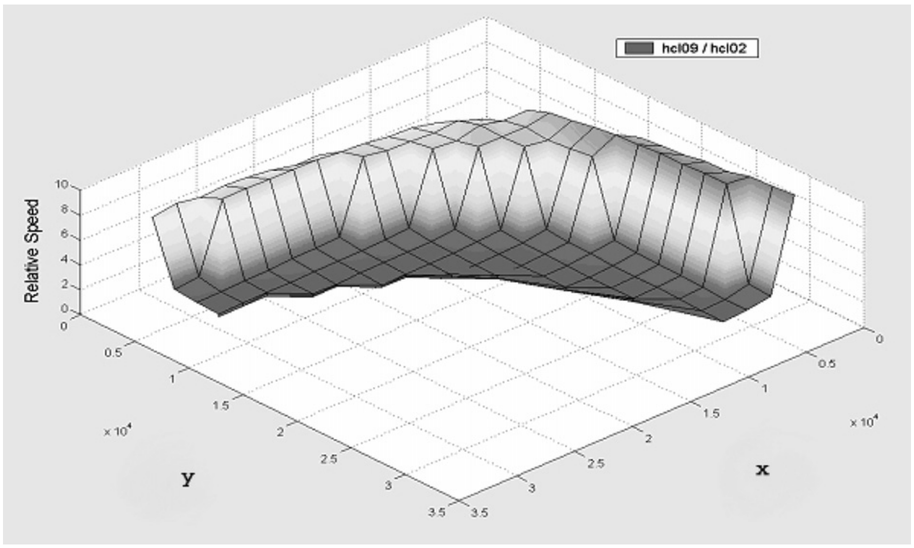

Fig. 4. The relative speed of two computers (‘hcl09’, ‘hcl02’) calculated as the ratio of their absolute speeds.

- $Proc(k,p,\Delta,w,d)$, a procedure searching for optimal allocation of a group of $w+1$ columns, $(k, k+1,\ldots,k+w)$, given columns $1,\ldots,k-1$ have been allocated and the total number of columns to be assigned to processor P_i is specified by the ith element of integer array Δ.

The output d is an integer array of size n, the ith element of which contains the processor to which the column i is assigned. The algorithm can be summarized as:

```
$(d_1,\ldots,d_n) = (0,\ldots,0)$;
$w = 0$;
$(n_1,\ldots,n_p) = \mathrm{HSPF}(p,n,S)$;
```

```
for ($k = 1$; $k < n$; $k = k + 1$) {
  $(n'_1, \ldots, n'_p) = \mathrm{HSPF}(p, n - k, S)$;
  if ($w{=}{=}0$)
  then if $((\exists! j \in [1,p])(n_j == n'_j + 1) \wedge (\forall i \neq j)(n_i == n'_i))$
    then {$d_k = j$; $(n_1, \ldots, n_p) = (n'_1, \ldots, n'_p)$;}
    else $w = 1$;
      else if $((\exists i \in [1,p])(n_i < n'_i))$
      then $w = w + 1$;
      else {
        for ($i = 1$; $i \leqslant p$; $i = i + 1$) {$\Delta_i = n_i - n'_i$; }
        $proc(k, p, \Delta, w, d)$;
        $(n_1, \ldots, n_p) = (n'_1, \ldots, n'_p)$;
        $w = 0$;
      }
}
if $((\exists i \in [1,p])(n_i = 1))$
then $d_n = i$;
```

Here, HSPF(p,m,S) returns the optimal distribution of a set of m equal elements over p heterogeneous processors $P_1,P_2,\ldots,P_p$ of respective speeds $S = \{s_1(m,y), s_2(m,y), \ldots, s_p(m,y)\}$ using the set-partitioning algorithm [13]. (HSPF stands for Heterogeneous Set Partitioning using the Functional model of processors). The distributed elements represent column panels of the $(m\times\ b)\times(m\times b)$ trailing matrix at step $(n - m)$ of the LU factorization. Function $f_i(y) = s_i(m,y)$ represents the speed of processor P_i depending on the number of column panels of the trailing matrix, y, updated by the processor at the step $(n - m)$ of the LU factorization. Fig. 5 gives geometrical interpretation of this step of the matrix-partitioning algorithm:

1. Surfaces $z_i = s_i(x,y)$ representing the absolute speeds of the processors are sectioned by the plane $x = n - k$ (as shown in Fig. 5(a) for three surfaces representing the absolute speeds of the processors ‘hcl02’, ‘hcl09’, ‘hcl11’ used in the experiments). A set of p curves on this plane (as shown in Fig. 5(b)) will represent the absolute speeds of the processors against variable y given parameter x is fixed.
2. The set-partitioning algorithm [13] is applied to this set of curves to obtain an optimal distribution of columns of the trailing matrix.

Proposition 3. *If assignment of a column is performed at each step of the algorithm, the FR algorithm returns the optimal allocation.*

Proof. If a column is assigned at each iteration of the FR algorithm, then the resulting allocation will be optimal by design. Indeed, in this case the distribution of columns over the processors will be produced by the HSPF and hence be optimal for each step of the LU factorization. □

Proposition 4. *If the speed of the processor is represented by a constant function of task size, the Functional Reverse algorithm returns the optimal allocation.*

Proof. If the speed of the processor is represented by a constant function of task size, the FR algorithm is functionally equivalent to the Reverse algorithm presented earlier. We have already proved that the Reverse algorithm returns the optimal allocation when constant performance model of heterogeneous processors is used. □

Proposition 5. *If assignment of a column panel is performed at each iteration of the main loop of the FR algorithm, its complexity will be bounded by* $O(p \times n \times \log_2 n)$.

Proof. At each iteration of this algorithm, we apply the HSPF. The first step of the HSPF, which involves intersection of p surfaces by a plane to produce p curves is of complexity $O(p)$. The second step of the HSPF is of complexity $O(p \times \log_2 n)$ [13]. Testing the condition $(\exists! j \in [1,p])(n_j == n'_j + 1) \wedge (\forall i \neq j)(n_i == n'_i)$ is of complexity $O(p)$. Since there are n such steps, the overall complexity of the algorithm is upper bounded by $n \times O(p \times \log_2 n) + n \times O(p) + n \times O(p) = O(p \times n \times \log_2 n)$. □

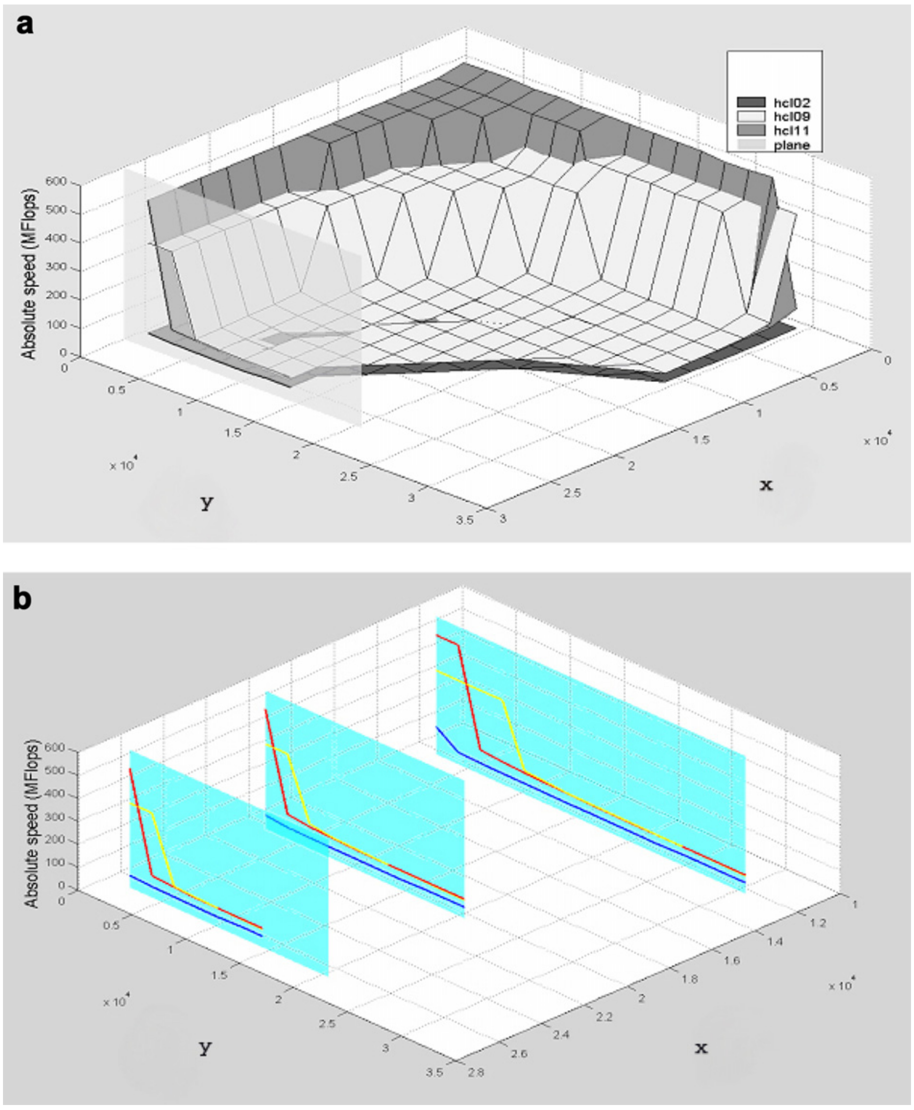

Fig. 5. (a) Three surfaces representing the absolute speeds of three processors ('hcl02', 'hcl09', 'hcl11') are sectioned by a plane $x = \text{const}$. (b) Curves on this plane represent the absolute speeds of the processors against variable y, given parameter x is fixed.

Heuristics. If the FR algorithm does not assign a column at each iteration of its main loop, then the optimality of the returned allocation is not guaranteed. The reason is that when we are forced to allocate a group of columns, $(k,k+1,\ldots,k+w)$, then even if procedure *Proc* finds a locally optimal allocation, minimizing the sum of the execution times of the steps $k,k+1,\ldots,k+w$ of the LU factorization (given columns $1,\ldots,k-1$ have been allocated), this allocation may not minimize the global execution time. Hence, sub-optimal allocations of the group of columns may be as good or even better as the exact optimal allocation. Therefore, in practice it does not make much sense to use an exact but exhaustive search algorithm in implementation of procedure *Proc*. Simple approximate algorithms of low complexity can return group allocations that are in average as good as exact optimal allocations.

The complexity of the FR algorithm depends on the complexity of procedure *proc*. This procedure determines the optimal order in which to assign the columns $\Delta_i(1 \leqslant i \leqslant p)$ to the processors. At each iteration of this algorithm, we apply the HSPF, which introduces the complexity $O(p \times \log_2 n) + O(p)$. Testing the condition $(\exists! j \in [1,p])(n_j == n'_j + 1) \wedge (\forall i \neq j)(n_i == n'_i)$ is of complexity O(p). Testing the condition $(\exists i \in [1,p])(n_i < n'_i)$ is also of complexity O(p). Plus at one or more iterations of the algorithm, there is an application of the heuristic procedure *proc*. Since there are n steps of the algorithm, the overall complexity of the algorithm is upper bounded by $n \times \mathrm{O}(p \times \log_2 n) + n \times \mathrm{O}(p) + n \times \mathrm{O}(p) + n \times \mathrm{O}(p) + n \times C_{\mathrm{h}} = \mathrm{O}(p \times n \times \log_2 n) + n \times C_{\mathrm{h}}$ where C_{h} is the complexity introduced by the heuristic functional procedure at each step.

Therefore if we do not want to drastically affect the overall complexity of the algorithm, efficient polynomial heuristics should be employed. Some of these heuristics are:

- *cycl:* Allocate cyclically one by one the column until all the Δ_i $(1 \leqslant i \leqslant p)$ have been assigned;
- *ftos:* Allocate from the most loaded (maximum Δ) to the least loaded (minimum Δ). Sort $\Delta_i (1 \leqslant i \leqslant p)$ in decreasing order. Assign Δ_i $(1 \leqslant i \leqslant p)$ number of column to each processor P_i in that order;
- *stof:* Allocate from the least loaded (minimum Δ) to the most loaded (maximum Δ). Sort Δ_i in increasing order. Assign Δ_i $(1 \leqslant i \leqslant p)$ number of column to each processor P_i in that order;
- *dflt:* Assign Δ_i $(1 \leqslant i \leqslant p)$ number of column to each processor P_i in the default order of processors input to this algorithm.

In Table 4, we demonstrate the heuristics for $n = 10$. The first column represents the step k of the algorithm. The second column shows the distributions obtained during step S2 by the set partitioning algorithm. The entry "Allocation made" denotes the rank of the processor to which the column k is assigned. At steps $k = 4$ and $k = 5$, the algorithm does not make any assignments. At $k = 6$, the different heuristics are applied to assign (2, 1) number of blocks to processors (P_1, P_3), respectively. Note here the heuristic *dflt* performs the same assignment as heuristic *ftos* but it may not be the case always. For example using the heuristic *cycl*, the final output d would be $(P_1 P_1 P_1 P_1 P_3 P_1 P_2 P_1 P_2 P_3)$.

Table 5 presents the complexities introduced by each of these heuristics. It can be seen that these heuristics do not affect the overall complexity of the algorithm, which is $\mathrm{O}(p \times n \times \log_2 n)$.

5.3. Functional GB algorithm

The GB algorithm presented earlier is extended to use the functional model of heterogeneous processors. The efficiency of the Functional Reverse algorithm over this algorithm is compared in the section on experimental results.

The Functional GB algorithm (FGB). This algorithm extends the Group Block algorithm presented earlier by using the functional model of heterogeneous processors. The main idea here is that at each step, the number

Table 4
Functional Reverse algorithm using heuristics

Step of the algorithm (k)	Distributions at step k			Allocation made			
	P_1	P_2	P_3				
	6	2	2				
1	5	2	2	P_1			
2	4	2	2	P_1			
3	3	2	2	P_1			
4	1	3	2	No allocation			
5	1	3	1	No allocation			
6	1	2	1	Heuristics			
				cycl	*ftos*	*stof*	*dflt*
				P_1, P_3, P_1	P_1, P_1, P_3	P_3, P_1, P_1	P_1, P_1, P_3
7	1	1	1	P_2			
8	0	1	1	P_1			
9	0	0	1	P_2			
10				P_3			

Table 5
Heuristics and their complexities

Heuristic	Complexity (C_h)
cycl	$O(p)$
ftos	$O(p \times \log_2 p)$
stof	$O(p \times \log_2 p)$
dflt	$O(p)$

of column panels per group and the distribution of the column panels in the group over the processors are calculated based on the absolute speeds of the processors given by the functional model, which are based on the size of the problem solved at that step. That is the number of column panels per group and the distribution of column panels in a group amongst the processors are variable.

The inputs to the algorithm are p, the number of heterogeneous processors in the one-dimensional arrangement, b, the block size, n, the size of the matrix in number of blocks of size $b \times b$ or the number of column panels, and $S = \{s_1(x,y), s_2(x,y), \ldots, s_p(x,y)\}$, the speed functions of the processors. The outputs are $G = \{g_1, \ldots, g_m\}$, an integer array of size m, the ith element of which contains the size of the group and d, an integer array of size $m \times p$ logically representing an array of shape $[m][p]$, the (i,j)th element of which contains the number of column panels in the group i assigned to processor j. The algorithm can be summarized as follows:

(1) The size g_1 of the first group of blocks is calculated as follows:
  (a) Apply HSPF(p,n,S) to return the optimal distribution $(n_1, \ldots, n_p)$ of n column panels where $\sum_{i=1}^{p} n_i = n$. Calculate the load index $l_i = n_i / \sum_{k=1}^{p} n_k$ $(1 \leqslant i \leqslant p)$.
  (b) The size of the group g_1 is equal to $\lfloor 1/\min(l_i) \rfloor$ $(1 \leqslant i \leqslant p)$. If $g_1/p < 2$, then $g_1 = \lfloor 2/\min(l_i) \rfloor$. This condition is imposed to ensure there is sufficient number of blocks in the group.
  (c) This group is now partitioned such that the number of column panels $d_1^{(i)}$ assigned to processor i in the group is proportional to the load indices l_i where $\sum_{i=1}^{p} d_1^{(i)} = g_1$ $(1 \leqslant i \leqslant p)$.

(2) To calculate the size g_2 of the second group, we repeat step 1 for the number of column panels equal to $n - g_1$ in matrix A. We recursively apply this procedure until we have fully vertically partitioned the matrix A.

(3) In each group, the processors are reordered to start from the slowest processors to the fastest processors for load balance purposes.

To calculate the complexity of the algorithm, consider the first step of the algorithm. The complexity of HSPF is $O(p \times \log_2 n)$ [13]. The complexity of step 1.b is $O(1)$. The complexity of step 1.c is $O(p \times \log_2 p)$. So the total complexity for this step is $O(p \times \log_2 n) + O(p \times \log_2 p) + O(1)$. In the worst case scenario, there are n possible groups. So the complexity of the algorithm is upper bounded by $n \times (O(p \times \log_2 n) + O(p \times \log_2 p) + O(1)) = O(p \times n \times \log_2 n)$.

5.4. Functional DP algorithm

The DP algorithm presented earlier is extended to use the functional model of heterogeneous processors. The efficiency of the Functional Reverse algorithm over this algorithm is compared in the section on experimental results.

The Functional DP algorithm (FDP). This algorithm extends the DP algorithm presented earlier by using the functional model of heterogeneous processors. The inputs to the algorithm are p, the number of heterogeneous processors in the one-dimensional arrangement, b, the block size, n, the size of the matrix in number of blocks of size $b \times b$ or the number of column panels, and $S = \{s_1(x,y), s_2(x,y), \ldots, s_p(x,y)\}$, the speed functions of the processors. The outputs are c, an integer array of size p, the ith element of which contains the number of column panels assigned to processor i, and d, an integer array of size n, the ith element of which contains the processor to which the column panel i is assigned. The algorithm can be summarized as follows:

```
$(c_1,\ldots,c_p)=(0,\ldots,0)$;
$(d_1,\ldots,d_n)=(0,\ldots,0)$;
for($k=1$; $k\leqslant n$; $k=k+1$) {
    $\mathrm{Cost}_{\min}=\infty$;
    $(n_1,\ldots,n_p)=\mathrm{HSPF}(p,k,S)$;
    for($i=1$; $i< =p$; $i$++) {
        $\mathrm{Cost}=(c_i+1)/n_i$;
        if ($\mathrm{Cost}<\mathrm{Cost}_{\min}$) {$\mathrm{Cost}_{\min}=\mathrm{Cost}$; $j=i$;}
    }
    $d_{n-k+1}=j$;
    $c_j=c_j+1$;
}
```

To calculate the complexity of the algorithm, consider an iteration of the algorithm. The complexity of HSPF is $O(p\times\log_2 n)$ [13]. The complexity of the for loop is $O(p)$. Therefore, the total complexity will be $O(p\times\log_2 n)+O(p)$. Since there are n iterations, the complexity of the algorithm is upper bounded by $n\times(O(p\times\log_2 n)+O(p))=O(p\times n\times\log_2 n)$.

5.5. Concluding remarks

In the general case, the data distribution algorithms employing the functional model of heterogeneous processors do not provide optimal solutions to the optimization problem presented at the beginning of Section 5.1. At the same time, the FR and the FDP algorithms will always return optimal solutions in the following two cases:

(a) The speeds of the processors are constant functions of the task size. In this case, the FDP algorithm is identical to the DP algorithm.
(b) The speeds of the processors allow the FR algorithm to assign a column at each step (see behind the Functional Reverse distribution algorithm Proposition 3). In this case, both the FR and the FDP algorithms return the optimal solution.

The FGB algorithm may return non-optimal solutions even in these two cases.

6. Experimental results

A small heterogeneous local network of sixteen different Linux workstations shown in Table 6 is used in the experiments. The network is based on 2 Gbit Ethernet with a switch enabling parallel communications between the computers. The experimental results are divided into two sections. The first section is devoted to building the functional representation of the speed of a processor. Then we present the experimental results on the local network of heterogeneous computers shown in Table 6.

6.1. Implementation of the functional algorithms

In heterogeneous environments, most processors experience constant and unpredictable fluctuations in the workload. Therefore, their performance is represented by speed bands rather than speed functions. This property of real-life environments can be used to minimize the cost of experimental building of the functional performance model of the processors. If we accept any appropriate function (continuous, etc.) fitting into the speed band as a satisfactory approximation of the speed function, then the problem of efficient building the functional performance model can be formulated as follows:

- Find the optimal set of task sizes such that:
  - Running the application for this set is enough to build an approximation of the speed function fitting into the speed band.
  - The total execution time to run the application for the set of task sizes is minimal.

Table 6
Specifications of 16 Linux computers of a heterogeneous network

Computer	GHz CPU	RAM (mBytes)	Cache (kBytes)	Functional model building time (s)
hcl01	3.6 Xeon	256	2048	565
hcl02	3.6 Xeon	256	2048	555
hcl03	3.4 Xeon	1024	1024	606
hcl04	3.4 Xeon	1024	1024	606
hcl05	3.4 Xeon	1024	1024	606
hcl06	3.4 Xeon	1024	1024	606
hcl07	3.4 Xeon	256	1024	480
hcl08	3.4 Xeon	256	1024	480
hcl09	1.8 AMD Opteron	1024	1024	550
hcl10	1.8 AMD Opteron	1024	1024	600
hcl11	3.2 P4	512	1024	425
hcl12	3.4 P4	512	1024	630
hcl13	2.9 Celeron	1024	256	445
hcl14	3.4 Xeon	1024	1024	485
hcl15	2.8 Xeon	1024	1024	485
hcl16	3.6 Xeon	1024	2048	485

One possible approach is to look for an optimal piecewise linear approximation of the speed function. A practical procedure to build a piecewise linear function fitting into the speed band of a processor is given in [20]. In brief, the Geometric Bisection Building Procedure (GBBP) presented in [20] exploits historic records of workload fluctuations of the processor in order to minimize the number of experimental points needed to accurately approximate the speed function by a piecewise linear function fitting into the band. For the experiments in this paper, the piecewise linear approximation of the speed band of the processor is built using a set of experimentally obtained points $(x,y, s(x,y))$ for different task sizes (x,y) where $s(x,y)$ is the speed exposed by the application when solving the task of size (x,y).

To obtain an experimental point for a task size (x,y), we execute the level-3 BLAS DGEMM routine [21] that is used in the LU factorization application to locally update a dense matrix of size $x \times y$. The total number of computations involved in updating $(\widetilde{A_{22}} \leftarrow A_{22}L_{21}U_{12})$ of the $x \times y$ matrix A_{22}, where L_{21} is a column of the size x and U_{12} is a row of the size y, will be $x \times y$ (remember that, generally speaking, the matrix elements represent $b \times b$ matrix blocks). The block size b used in the experiments is 32, which is typical for cache-based workstations [18,19]. Therefore, the absolute speed of the processor $s(x, y)$ can be calculated as $x \times y$ divided by the execution time of the application. The piecewise linear approximation is obtained by connecting these experimental points. This gives us a function, f: $\mathbf{N}^2 \rightarrow \mathbf{R}_+$, mapping task sizes to speeds of the processor. The functional performance model of the processor is obtained by continuous extension of function f: $\mathbf{N}^2 \rightarrow \mathbf{R}_+$ to function g: $\mathbf{R}_+^2 \rightarrow \mathbf{R}_+$. The speed function is geometrically represented by a surface as shown in Fig. 3 for one of the computers 'hcl11' used in the experiments. Fig. 4 shows the geometrical representation of the relative speed of these two processors calculated as the ratio of their absolute speeds. One can see that the relative speed varies significantly depending on the value of variables x and y.

The application of the set-partitioning algorithm HSPF at step k of the FR algorithm can be summarized as follows. When partitioning the trailing square $(n - k) \times (n - k)$ matrix, we use the fact that the height of the partitions is fixed and equal to $n - k$. Firstly, we section the surfaces representing the absolute speeds of the processors by the plane $x = n - k$. This is illustrated in Fig. 5(a) for three surfaces representing the absolute speeds of the processors 'hcl02', 'hcl09' and 'hcl11' used in the experiments. This way we obtain a set of p curves on this plane that represent the absolute speeds of the p processors against parameter y given parameter $x = n - k$ is fixed. Then, the set-partitioning algorithm [13] is applied to this set of p curves to obtain optimal distribution of the trailing square $(n - k) \times (n - k)$ matrix.

The last column in Table 6 shows the execution times taken to build the functional performance model of each processor used in the experiments. This table demonstrates that building the functional performance model is inexpensive compared to the execution times of the parallel LU factorization, which range from minutes to hours. It should be noted that the building of the functional performance model could be performed in parallel for each processor in the network shown in Table 6.

6.2. Numerical results

Fig. 6 shows the first set of experiments. For the range of task sizes (1024–11,264) used in these experiments, the speed of the processor is a constant function of the task size. These experiments demonstrate the optimality of the FR and the DP algorithms over the GB algorithm when the speed of the processor is a constant function of the task size. It should be noted that:

FDP algorithm is functionally equivalent to the DP algorithm when the speed of the processor is represented by a constant function of task size. The figure shows the execution times of the LU factorization application using these algorithms. The single number speeds of the processors used for these experiments are obtained by running the DGEMM routine to update a dense non-square matrix of size 5120×320. The ratio of speeds of the most powerful computer *hcl16* and the least powerful computer *hcl01* is $609/226 \approx 2.7$.

Table 7 shows the second set of experiments showing the execution times of the different data distribution algorithms presented in this paper. We consider two cases for comparison in the range (1024, 25,600) of matrix sizes. The GB and DP algorithms uses single number speeds. For the first case the single number speeds are obtained by running the DGEMM routine to update a dense non-square matrix of size $16{,}384 \times 1024$. This case covers the range of small sized matrices. For the second case the single number speeds are obtained by running the DGEMM routine to update a dense non-square matrix of size $20{,}480 \times 1440$. This case covers the range of large sized matrices. The ratios of speeds of the most powerful computer *hcl16* and the least powerful computer *hcl01* in these cases are ($531/131 = 4.4$) and ($579/64 = 9$), respectively. The *cycl* heuristic has been used in the FR algorithm.

Table 8 shows the execution times of the LU factorization application employing different heuristics in the FR algorithm. The heuristics *ftos* and *cycl* outperform the heuristics *stof* and *dflt*. It is also shown that these heuristics perform no worse than the heuristic employing an exhaustive search algorithm.

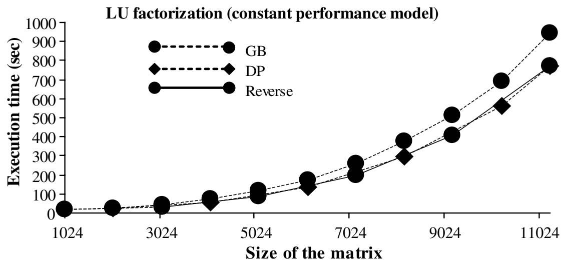

Fig. 6. Execution times of the FR, DP, and GB distribution strategies for LU factorization of a dense square matrix.

Table 7
Execution times (in seconds) of the LU factorization using different data distribution algorithms

Size of the matrix	FR	FDP	FGB	Reverse/DP		GB	
				$16{,}384 \times 1024$	$20{,}480 \times 1440$	$16{,}384 \times 1024$	$20{,}480 \times 1440$
1024	**15**	17	18	16	18	20	18
5120	**86**	155	119	103	109	138	155
10,240	**564**	1228	690	668	711	919	926
15,360	**2244**	3584	2918	2665	2863	2829	3018
20,480	**7014**	10,801	8908	9014	9054	9188	9213
25,360	**14,279**	22,418	19,505	27,204	26,784	27,508	26,983

Table 8
Execution times (in seconds) of the LU factorization employing the FR algorithm using the heuristics

Size of the matrix	*ftos*	*cycl*	*stof*	*dflt*	Exhaustive search
11,264	866	**790**	818	826	883
12,288	1360	**1283**	1408	1377	1467
13,312	1657	**1575**	1706	1673	1801
14,336	1905	**1854**	1977	1955	2168
15,360	2340	**2244**	2371	2383	2585
16,384	3000	**2871**	3120	3095	3400
17,408	3810	**3464**	4003	4001	4250
18,432	4908	**4501**	5217	5293	5699
19,456	**6164**	6287	6943	6943	6503
20,480	7439	**7014**	7837	7598	7659
21,504	8440	**8279**	9022	9034	8891
22,528	**10,854**	10,939	12,424	12,392	12,307
23,552	**12,175**	13,048	14,052	14,515	12,816
24,576	**13,281**	13,314	15,385	15,825	14,221
25,600	14,780	**14,279**	16,838	16,552	16,731

The last column in the table shows the execution times of the LU factorization application for the locally optimal mapping obtained by exhaustive search. Unlike results for the heuristics, in this case, the execution time of the LU factorization application does not include the execution time of the exhaustive search because of its very high cost.

Results for task sizes between 1024 and 10,240 are not shown in the table. This is because for this range, the speed of the processor is a constant function of the task size and the heuristics do not come into play.

It can be seen that the FR algorithm employing the functional model of heterogeneous processors performs well for all sizes of matrices. Firstly, why do the Reverse and the DP algorithms perform better than the GB algorithm when the speed of the processor is represented by a constant function of the task size? The main reason is that the GB algorithm imposes additional restrictions on the mapping of the columns to the processors. These restrictions are that the matrix is partitioned into groups, all of which have the same number of blocks. The number of columns per group (size of the group) and the distribution of the columns in the group over the processors are fixed. The Reverse and the DP algorithms impose no such limitations on the mapping.

Why does the FR algorithm outperform the FDP algorithm when the speed of the processor is a function of the task size? First of all, if an assignment of the column is made at each step, both the FR and the FDP algorithms give the optimal solution. It can be proven that the FDP algorithm provides the same mapping of column panels as the FR algorithm in this case. If an assignment of the column panel can not be made at each step, the FR algorithm performs better. The main reason is that the FDP algorithm will keep assigning a column panel at each step even in this case, meanwhile the FR algorithm postpones the mapping decision until it has enough information to better map the group of column panels to the processors.

7. Conclusions and future work

In this paper, we presented static data distribution algorithms 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.

In the general case, the data distribution algorithms employing the functional model of heterogeneous processors do not provide optimal solutions to the problem of distributing a large dense square matrix for its parallel LU factorization over a one-dimensional arrangement of heterogeneous processors using their functional performance model. It is shown that the FR algorithm employing the functional model of heterogeneous processors performs well for all sizes of matrices. Of all the presented efficient polynomial heuristics that can be employed in the FR algorithm, the heuristics *ftos* and *cycl* perform the best. It is also shown that these heuristics perform no worse than the exhaustive search considering all the possible mappings.

Future work would involve extension of the distribution algorithms for two dimensional processor grids. For two-dimensional processor grids, the block cyclic distribution as used in ScaLAPACK is more natural and scalable. However the problem of data partitioning employing the functional model of heterogeneous processors and the block cyclic distribution is very complex and is open for research. This can be deduced from the complexity of the problem of cyclic distribution of columns over a one-dimensional arrangement of heterogeneous processors demonstrated in this paper. We can speculate that the FR algorithm can be applied along the row and the column dimensions of the matrix for data distribution on two-dimensional arrangement of heterogeneous processors. But a cursory study shows that this is not trivial.

References

[1] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, Experiments with mpC: efficient solving regular problems on heterogeneous networks of computers via irregularization, in: Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98), Lecture Notes in Computer Science, vol. 1457, 1998, pp. 332–343.
[2] A. Kalinov, A. Lastovetsky, Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers, in: Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99), Lecture Notes in Computer Science, vol. 1593, 1999, pp. 191–200.
[3] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, Y. Robert, A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers), IEEE Transactions on Computers 50 (10) (2001) 1052–1070, October.
[4] P. Boulet, J. Dongarra, F. Rastello, Y. Robert, F. Vivien, Algorithmic issues on heterogeneous computing platforms, Parallel Processing Letters 9 (2) (1999) 197–213.
[5] J. Barbosa, J. Tavares, A.J. Padilha, Linear algebra algorithms in a heterogeneous cluster of personal computers, in: Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000), Cancun, Mexico, IEEE Computer Society Press, 2000, pp. 147–159, May.
[6] J. Barbosa, C.N. Morais, A.J. Padilha, Simulation of data distribution strategies for LU factorization on heterogeneous machines, in: Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2004), 26–30 April 2004, Santa Fe, New Mexico, USA, CD-ROM/Abstracts Proceedings, IEEE Computer Society, 2004.
[7] X. Du, X. Zhang, Z. Zhu, Memory hierarchy considerations for cost-effective cluster computing, IEEE Transactions on Computers 49 (9) (2000) 915–933.
[8] S. Manegold, P. Boncz, M. Kersten, Generic database cost models for hierarchical memory systems, in: Proceedings of International Conference on Very Large Data Bases (VLDB), August 20–23, 2002, pp. 191–202.
[9] Yi-Min Wang, Memory latency consideration for load sharing on heterogeneous network of workstations, Journal of Systems Architecture 52 (1) (2006) 13–24.
[10] T. Rauber, G. Runger, A hierarchical computation model for distributed shared-memory machines, in: Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing (PDP'01), IEEE Computer Society, 2001, pp. 57–64.
[11] M. Drozdowski, P. Wolniewicz, Out-of-core divisible load processing, IEEE Transactions on Parallel and Distributed Systems 14 (10) (2003) 1048–1056, October.
[12] A. Lastovetsky, R. Reddy, Data partitioning with a realistic performance model of networks of heterogeneous computers, in: Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2004), 26–30 April 2004, Santa Fe, New Mexico, USA, CD-ROM/Abstracts Proceedings, IEEE Computer Society, 2004.
[13] A. Lastovetsky, R. Reddy, Data partitioning with a functional performance model of heterogeneous processors, International Journal of High Performance Computing Applications 21 (1) (2007) 76–90.
[14] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin-Cummings, Addison-Wesley, 1994, 597 p.
[15] C.A. Kitchen, M.F. Guest, M.J. Deegan, I. Kozin, R. Wain, The euroben benchmarks 2. Probing Parallel Performance on Xeon, Opteron, Itanium2 and Power5-based Clusters, March 2006. Available from: <http://www.cse.clrc.ac.uk/disco/DLAB_BENCH_WEB/Euroben2.0-dm/euroben2.0-dm.shtml>.
[16] A.J. van der Steen, The benchmark of the EuroBen group, Parallel Computing 17 (10–11) (1991) 1211–1221.
[17] A. Lastovetsky, R. Reddy, Data partitioning for multiprocessors with memory heterogeneity and memory constraints, Scientific Programming 13 (2) (2005) 93–112.
[18] J. Choi, J. Dongarra, L.S. Ostrouchov, A.P. Petitet, D.W. Walker, R.C. Whaley, The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines, Scientific Programming, 1058-9244 5 (3) (1996) 173–184.
[19] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.

[20] A. Lastovetsky, R. Reddy, R. Higgins, Building the functional performance model of a processor, in: Proceedings of the 21st Annual ACM Symposium on Applied Computing (ACM SAC 2006), April 23–24, Dijon, France, 2006.

[21] J. Dongarra, J.D. Croz, I.S. Duff, S. Hammarling, A set of level-3 basic linear algebra subprograms, ACM Transactions on Mathematical Software 16 (1) (1990) 1–17, March.