A Novel Algorithm of Optimal Matrix Partitioning for Parallel Dense Factorization on Heterogeneous Processors

Alexey Lastovetsky and Ravi Reddy

School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland
`{alexey.lastovetsky,manumachu.reddy}@ucd.ie`

Abstract. In this paper, we present a novel algorithm of optimal matrix partitioning for parallel dense matrix factorization on heterogeneous processors based on their constant performance model. We prove the correctness of the algorithm and estimate its complexity. We demonstrate that this algorithm better suits extensions to more complicated, non-constant, performance models of heterogeneous processors than traditional algorithms.

1 Introduction

The paper presents a novel algorithm of optimal matrix partitioning for parallel dense matrix factorization on heterogeneous processors based on their constant performance model. We prove the correctness of the algorithm and estimate its complexity. We demonstrate that this algorithm better suits extensions to more complicated, non-constant, performance models of heterogeneous processors, such as a model presented in [1,2], than traditional algorithms.

A number of matrix distribution strategies for parallel dense matrix factorization in heterogeneous environments have been designed and implemented. Arapov *et al.*, [3] 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 [4]. Beaumont *et al.*, [5-6] employ a dynamic programming algorithm (DP) to partition the matrix in parallel 1D LU decomposition. When processor speeds are accurately known and guaranteed not to change during program execution, the dynamic programming algorithm provides the best possible load balancing of the processors. A static group block distribution strategy [7-8] is used in parallel 1D LU decomposition to partition the matrix into groups (or *generalized blocks* in terms of [4]), 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.

We present in this paper a novel matrix partitioning algorithm for 1D LU decomposition called the Reverse algorithm. Like the DP algorithm, the Reverse algorithm always returns an optimal solution. The complexity of the Reverse algorithm is a bit worse than that of the DP algorithm, but the algorithm has one important advantage. It better suits extensions to more complicated, non-constant, performance models of heterogeneous processors, such as the functional performance model [1,2], than traditional algorithms.

The rest of the paper is organized as follows. In Section 2, we present the homogeneous LU factorization algorithm that is used for our heterogeneous modification. In section 3, we outline two existing heterogeneous modifications of this algorithm using the constant model of heterogeneous processors before presenting our original modification, the Reverse algorithm. This section also presents the correctness of the algorithm and its complexity. Finally we present experimental results on a local network of heterogeneous processors to demonstrate why the proposed algorithm better suits extensions to the functional performance model of heterogeneous processors than the traditional algorithms.

2 LU Factorization on Homogeneous Multiprocessors

Before we present our matrix partitioning algorithm, 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 Figure 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 [9,10].

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

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

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

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

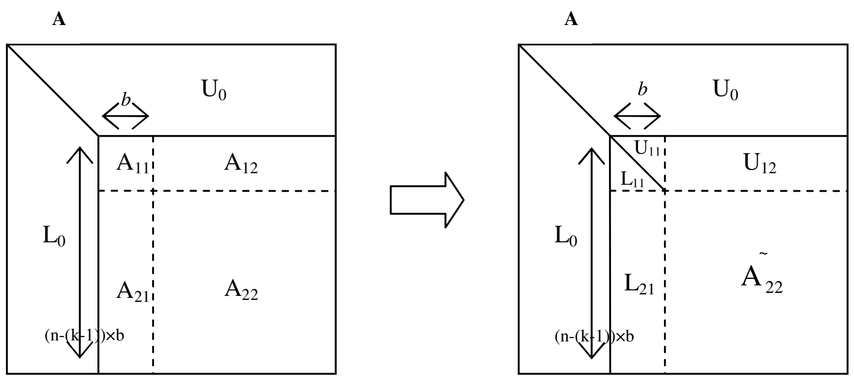

Fig. 1. One step of the LU factorization algorithm of a dense matrix A of size (n×b)×(n×b)

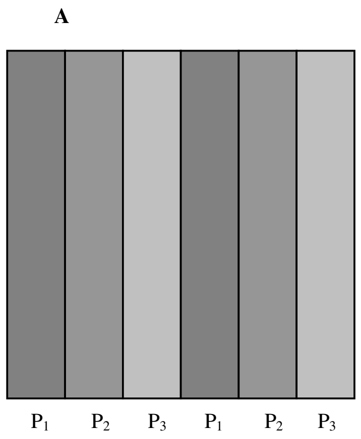

Fig. 2. Column-oriented CYCLIC distribution of six column blocks on a one-dimensional array of three homogeneous processors

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

$$U_{12} \leftarrow (L_{11})^{-1} A_{12},$$

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22}.$$

The LU factorization can be done by recursively applying the steps outlined above to the $(m\text{-}b)\times(m\text{-}b)$ matrix $\tilde{A}_{22}$. 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. 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 Figure 2. The cyclic distribution assigns columns of blocks with numbers 1,2,…,n to processors 1,2,…,p,1,2,…,p,1,2,…, respectively, for a p-processor linear array (n»p), until all n columns of blocks are assigned.
2. The algorithm consists of n steps. At each step (k=1,2,...),

– 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 [10] 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,…,b)]

- **PDAMAX:** find the (absolute) maximum element of the i-th column and its location
- **PDSWAP:** interchange the i-th row with the row that holds the maximum
- **PDSCAL:** scale the i-th 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.

3 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 2. 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 $(n»p)$ heterogeneous processors P_1, P_2, ..., P_p of relative speeds $S=\{s_1, s_2, ..., s_p\}$, $\sum_{i=1}^{p} s_i = 1$, 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, $s_i = \frac{a_i}{\sum_{i=1}^{p} a_i}$. 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 $\max_i \frac{n_i^{(k)}}{s_i}$ 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 $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}}$, 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 $\max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}}$ for each k, it will also minimize $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}}$. Because

$\max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}} = \max_{i=1}^{p} \frac{n_i^{(k)}}{s_i \times \sum_{i=1}^{p} a_i^{(k)}} = \frac{1}{\sum_{i=1}^{p} a_i^{(k)}} \times \max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$, then for any given k the problem of minimization of $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}}$ will be equivalent to the problem of minimization of $\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$. Therefore, if we are lucky and there exists an allocation that minimizes $\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$ for each step k of the LU factorization, then the allocation will be globally optimal, minimizing $\sum_{k=1}^{n} \max_{i=1}^{p} \frac{n_i^{(k)}}{a_i^{(k)}}$. Fortunately, such an allocation does exist [5,6].

Now we briefly outline two existing approaches to solve the above distribution problem, which are the Group Block (GB) distribution algorithm [7] and the Dynamic Programming (DP) distribution algorithm [5,6].

The GB algorithm. This algorithm partitions the matrix into groups (or *generalized blocks* in terms of [4]), 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 $S=\{s_1, s_2, ..., s_p\}$ ($\sum_{i=1}^{p} s_i = 1$), the relative speeds of the processors. The outputs are g, the size of the group, and d, an integer array of size p, the i-th 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 $\lfloor 1/\min(s_i) \rfloor$ ($1 \le i \le p$). 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 $S=\{s_1, s_2, ..., s_p\}$($\sum_{i=1}^{p} s_i = 1$), the relative speeds of the processors. The outputs are c, an integer array of size p, the i-th element of which contains the number of column panels assigned to processor i, and d, an integer array of size n, the i-th 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; $k \le n$; $k$=$k$+1) {
        Cost$_{min}$=$\infty$;
        for($i$=1; $i$<=$p$; $i$=$i$+1) {
            Cost=($c_i$+1)/$s_i$;
            if (Cost < Cost$_{min}$) {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 [6]. 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 better suits extensions to more complicated, non-constant, performance models of heterogeneous processors (such as the functional model [1, 2]) than both the DP and GB algorithms.

The Reverse algorithm. This algorithm generates the optimal distribution $(n_1^{(k)},\ldots,n_p^{(k)})$ of n×b column panels of the dense (n×b)×(n×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,…,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×b or the number of column panels, and $S=\{s_1, s_2, ..., s_p\}$($\sum_{i=1}^{p} s_i = 1$), the relative speeds of the processors. The output is d, an integer array of size n, the i-th element of which contains the processor to which the column panel i is assigned. The algorithm can be summarized as follows:

```
$(d_1,\ldots,d_n)=(0,\ldots,0)$;
$w=0$;
$(n_1,\ldots,n_p)=HSP(p, n, S)$;
for ($k=1$; $k<n$; $k=k+1$) {
  $(n'_1,\ldots,n'_p) = HSP(p, n-k, S)$;
  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 ( $(\exists\, i \in [1,p])(n_i < n'_i)$ )
      then $w=w+1$;
      else {
        for ($i=1$; $i\le p$; $i=i+1$)
          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 ( $(\exists\, i \in [1,p])(n_i == 1)$ )
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 $S=\{s_1, s_2, ..., s_p\}$ by applying the algorithm for optimal distribution of independent chunks of computations from [5]

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

(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

Table 2. Distribution algorithms and their complexities

Distribution Algorithm	**Complexity**
GB	$O(p \times \log_2 p)$
DP	$O(p \times n)$
Reverse	$O(p \times n \times \log_2 p)$

processor P_2 is allocated column panel 6. The output d in this case would be $(P_1P_1P_1P_1P_1P_3P_2P_1P_2P_3)$.

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 $(n_1^{(k)},\ldots,n_p^{(k)})$, $(n_1^{(k+1)},\ldots,n_p^{(k+1)})$, …, $(n_1^{(k+w)},\ldots,n_p^{(k+w)})$ such that

– the distributions are optimal for steps k, k+1,…, 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 $(\forall i)(a_i \ge b_i) \wedge (\exists i)(a_i > b_i)$).

Lemma 1. Let $(n_1,\ldots,n_p)$ and $(n_1^{'},\ldots,n_p^{'})$ be optimal distributions such that

$$n = \sum_{i=1}^{p} n_i > \sum_{i=1}^{p} n_i^{'} = n^{'}, \quad (\exists i)(n_i < n_i^{'}) \quad \text{and} \quad (\forall j)(\max_{i=1}^{p} \frac{n_i}{s_i} \le \frac{n_j+1}{s_j}).$$

Then, $\max_{i=1}^{p} \frac{n_i}{s_i} = \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$.

Proof of Lemma 1. As $n > n^{'}$ and $(n_1,\ldots,n_p)$ and $(n_1^{'},\ldots,n_p^{'})$ are both optimal distributions, then $\max_{i=1}^{p} \frac{n_i}{s_i} \ge \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$. On the other hand, there exists $j \in [1,p]$ such that $n_j < n_j^{'}$, which implies $n_j + 1 \le n_j^{'}$. Therefore, $\max_{i=1}^{p} \frac{n_i^{'}}{s_i} \ge \frac{n_j^{'}}{s_j} \ge \frac{n_j+1}{s_j}$. As we assumed that $(\forall j)(\max_{i=1}^{p} \frac{n_i}{s_i} \le \frac{n_j+1}{s_j})$, then

$\max_{i=1}^{p} \frac{n_i}{s_i} \le \frac{n_j+1}{s_j} \le \frac{n_j^{'}}{s_j} \le \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$. Thus, from $\max_{i=1}^{p} \frac{n_i}{s_i} \ge \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$ and $\max_{i=1}^{p} \frac{n_i}{s_i} \le \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$ we conclude that $\max_{i=1}^{p} \frac{n_i}{s_i} = \max_{i=1}^{p} \frac{n_i^{'}}{s_i}$. □

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 $(\forall j)(\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i} \le \frac{n_j^{(k)}+1}{s_j})$ (see [5,6]). Therefore, $\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i} = \max_{i=1}^{p} \frac{n_i^{(k+1)}}{s_i} = \ldots = \max_{i=1}^{p} \frac{n_i^{(k+w-1)}}{s_i}$. In particular, this means that for any $(m_1,\ldots,m_p)$ such that $\min_{j=k}^{k+w-1} n_i^{(j)} \le m_i \le \max_{j=k}^{k+w-1} n_i^{(j)}$ ($i = 1,\ldots,p$), we will have $\max_{i=1}^{p} \frac{m_i}{s_i} = \max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$. The allocations made in the end by the Reverse algorithm for the column panels *k*, *k*+1,…,*k*+*w*-1 result in a new sequence of distributions for steps *k*, *k*+1,…,*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 $\min_{j=k}^{k+w-1} n_i^{(j)} \le m_i \le \max_{j=k}^{k+w-1} n_i^{(j)}$ ($i = 1,\ldots,p$). Therefore, all they will have the same cost $\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$, 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 $O(p \times \log_2 p)$ [5]. Testing the condition $(\exists! j \in [1,p])(n_j == n_j^{'} + 1) \wedge (\forall i \ne j)(n_i == n_i^{'})$ 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≤p; i=i+1)
  for ($\Delta = n_i - n_i^{'}$; Δ≠0; Δ=Δ-1, w=w-1)
    $d_{k-w}$=i;
```

Table 3. Specifications of sixteen Linux computers of a heterogeneous network

Processor	GHz CPU	RAM (mBytes)	Cache (kBytes)	Absolute speed (MFlops)
hcl01	3.6 Xeon	256	2048	246
hcl02	3.6 Xeon	256	2048	226
hcl03	3.4 Xeon	1024	1024	258
hcl04	3.4 Xeon	1024	1024	258
hcl05	3.4 Xeon	1024	1024	260
hcl06	3.4 Xeon	1024	1024	258
hcl07	3.4 Xeon	256	1024	257
hcl08	3.4 Xeon	256	1024	257
hcl09	1.8 AMD Opteron	1024	1024	386
hcl10	1.8 AMD Opteron	1024	1024	347
hcl11	3.2 P4	512	1024	518
hcl12	3.4 P4	512	1024	258
hcl13	2.9 Celeron	1024	256	397
hcl14	3.4 Xeon	1024	1024	558
hcl15	2.8 Xeon	1024	1024	472
hcl16	3.6 Xeon	1024	2048	609

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 $n \times O(p \times \log_2 p) + n \times O(p) + n \times O(p) + p \times n \times O(1) = O(p \times n \times \log_2 p)$. Table 2 presents the complexities of the algorithms employing the constant performance model of heterogeneous processors.

4 Experimental Results

A small heterogeneous local network of sixteen different Linux workstations shown in Table 3 is used in the experiments. The network is based on 2 Gbit Ethernet with a switch enabling parallel communications between the computers.

The absolute speed of a processor is obtained by running the DGEMM routine that is used in our application to locally update a dense non-square matrix of size $n_1 \times n_2$. DGEMM is a level-3 BLAS routine [11] supplied by Automatically Tuned Linear Algebra Software (ATLAS) [12]. ATLAS is a package that generates efficient code for basic linear algebra operations. The total number of computations involved in updating $A_{22}=A_{22}-L_{21} \times U_{12}$ of the rectangular $n_1 \times n_2$ matrix A_{22}, where L_{21} is a matrix of the size $n_1 \times b$ and U_{12} is a matrix of the size $b \times n_2$, is $2 \times b \times n_1 \times n_2$. The block size b used in the experiments is 32, which is typical for cache-based workstations [9,10].

Figure 3 shows the first set of experiments. For the range of problem sizes used in these experiments, the speed of the processor is a constant function of the problem size. These experiments demonstrate the optimality of the Reverse and the DP

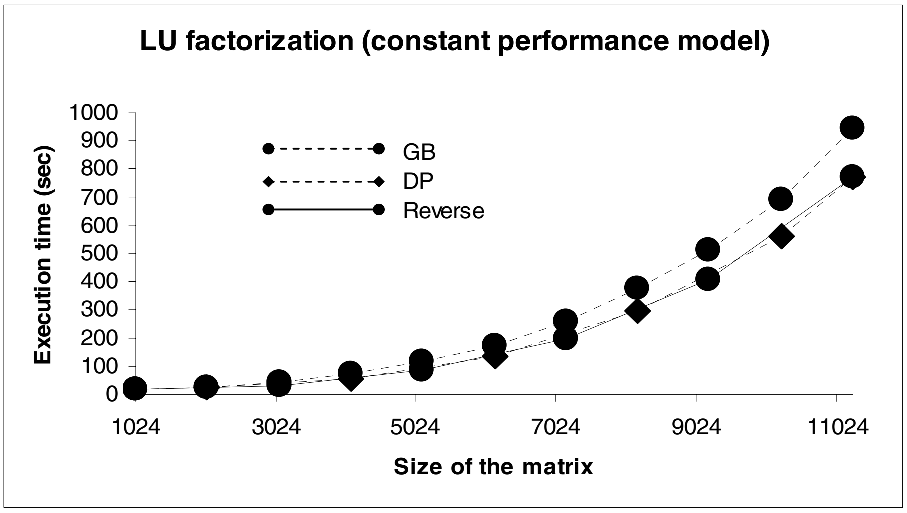

Fig. 3. Execution times of the Reverse, DP, and GB distribution strategies for LU decomposition of a dense square matrix

algorithms over the GB algorithm when the speed of the processor is a constant function of the problem 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. These speeds are shown in the last column of Table 3. The ratio of speeds of the most powerful computer *hcl16* and the least powerful computer *hcl01* is 609/226 ≈ 2.7.

Tables 4 and 5 show the second set of experiments showing the execution times of the different strategies presented in this paper along with their extensions using the functional model of heterogeneous processors [1, 2]. The strategies FDP, FGB, and FR are extensions of the DP, GB, and the Reverse algorithms respectively using the functional model of heterogeneous processors.

We consider two cases for comparison in the range (1024, 25600) 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 16384×1024. This case covers the range of small sized matrices. The results for this case are shown in Table 4. For the second case the single number speeds are obtained by running the DGEMM routine to update a dense non-square matrix of size 20480×1440. This case covers the range of large sized matrices. The results for this case are shown in Table 5. 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.

It can be seen that the FR algorithm, which is an extension of the Reverse algorithm and employing the functional model of heterogeneous processors performs well for all sizes of matrices. The Reverse and the DP algorithms perform better than the GB algorithm when the speed of the processor is represented by a constant

Table 4. Execution times (in seconds) of the LU factorization using different data distribution algorithms

Size of the matrix	**FR**	**FDP**	**FGB**	**Reverse/DP**	**GB**
1024	**15**	17	18	16	20
5120	**86**	155	119	103	138
10240	**564**	1228	690	668	919
15360	**2244**	3584	2918	2665	2829
20480	**7014**	10801	8908	9014	9188
25360	**14279**	22418	19505	27204	27508

Table 5. Execution times (in seconds) of the LU factorization using different data distribution algorithms

Size of the matrix	**FR**	**FDP**	**FGB**	**Reverse/DP**	**GB**
1024	**15**	17	18	18	18
5120	**86**	155	119	109	155
10240	**564**	1228	690	711	926
15360	**2244**	3584	2918	2863	3018
20480	**7014**	10801	8908	9054	9213
25360	**14279**	22418	19505	26784	26983

function of the problem 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.

5 Conclusions and Future Work

In this paper, we presented a novel algorithm of optimal matrix partitioning for parallel dense matrix factorization on heterogeneous processors based on their constant performance model. We prove the correctness of the algorithm and estimate its complexity. We demonstrate that this algorithm better suits extensions to more complicated, non-constant, performance models of heterogeneous processors than traditional algorithms.

Acknowledgement

This work was supported by the Science Foundation Ireland (SFI).

References

[1] Lastovetsky, A., Reddy, R.: Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers. In: Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04), IEEE Computer Society Press, Los Alamitos (2004)

[2] Lastovetsky, A., Reddy, R.: Data Partitioning with a Functional Performance Model of Heterogeneous Processors. International Journal of High Performance Computing Applications 21, 76–90 (2007)

[3] Arapov, D., Kalinov, A., Lastovetsky, A., Ledovskih, I.: Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization. In: Ferreira, A., Rolim, J.D.P., Teng, S.-H. (eds.) IRREGULAR 1998. LNCS, vol. 1457, pp. 332–343. Springer, Heidelberg (1998)

[4] Kalinov, A., Lastovetsky, A.: Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. Journal of Parallel and Distributed Computing 61, 520–535 (2001)

[5] Beaumont, O., Boudet, V., Petitet, A., Rastello, F., Robert, Y.: A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). IEEE Transactions on Computers 50, 1052–1070 (2001)

[6] Boulet, P., Dongarra, J., Rastello, F., Robert, Y., Vivien, F.: Algorithmic issues on heterogeneous computing platforms. Parallel Processing Letters 9, 197–213 (1999)

[7] Barbosa, J., Tavares, J., Padilha, A.J.: Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers. In: 9th Heterogeneous Computing Workshop (HCW 2000), pp. 147–159 (2000)

[8] Barbosa, J., Morais, C.N., Padilha, A.J.: Simulation of Data Distribution Strategies for LU Factorization on Heterogeneous Machines. In: Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), IEEE Computer Society Press, Los Alamitos (2003)

[9] 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 Routines. Scientific Programming 5, 173–184 (1996)

[10] Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.: ScaLAPACK User's Guide. SIAM (1997)

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

[12] Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the atlas project. Technical report, Department of Computer Sciences, University of Tennessee, Knoxville (2000)