11-1995

Using PASSION System on LU Factorization

Haluk Rahmi Topcuoglu
Syracuse University

Alok Choudhary
Syracuse University

SU-CIS-95-4

Using PASSION System on LU Factorization

Haluk Topcuoglu and Alok Choudhary

November, 1995

School of Computer and Information Science
Syracuse University
Suite 2-120, Center for Science and Technology
Syracuse, New York 13244-4100

Technical Report SU-CIS-95-4

Using PASSION System on LU Factorization

Haluk Topcuoglu and Alok Choudhary

School of Computer and Information Science

Syracuse University

Syracuse, NY 13244-4100

December 18, 1995

Abstract

Parallel I/O subsystems are added to massively parallel computers in order to lessen I/O bottleneck to some extent. Up to now, a few number of parallel software systems have been designed and implemented to assist programmers in I/O intensive applications; PASSION is one of them. By providing parallel I/O support at the language, compiler and run-time level, PASSION system explores the large design space of parallel systems. The target of this paper is to show the performance benefits of using PASSION I/O libraries at runtime in comparison with using conventional parallel I/O primitives for high performance parallel I/O in LU factorization kernel, a very widely used scientific kernel.

1 Introduction

Many supercomputing applications, like Grand Challenge problems, are extremely complex and require significant amount of processing time and data. Processing performance of multiprocessor systems has grown two or three orders of magnitude over the past decade and while memory density doubles every two years. However, technological advances on I/O can not catch the performance improvements of multiprocessor systems; therefore I/O system is a bottleneck in modern high performance systems and it limits the overall system performance.

Grand Challenge applications [1] require 500Mbytes to 500Gbytes of data storage. However, all the data required by these programs can not fit in main memory and needs to be stored on disks. These applications usually implemented in a way that at any time only a portion of it resides in memory (in-core) and the rest resides on secondary storages (out-of-core). Checkpointing is another reason that parallel programs require I/O. Applications that run for long hours may be stopped several times because of system failures or user requests. Storing the intermediate state at those points and restarting from the intermediate results require large volume of I/O.

The performance of I/O systems highly depends on data distribution and data management policies. Up to now a few number of systems have been designed and some of them have been implemented to assist programmers for I/O intensive applications. PASSION [2] (Parallel and Scalable Software for Input-Output) is one of these scarce systems that provides software support for I/O intensive out-of-core loosely synchronous problems. PASSION system provides software support for parallel I/O at the compiler, run-time and file system levels. This system also relieves the user from doing explicit low-level tedious work; user is only required to supply the portions of the file to be read or written. In order to show the performance benefits of the PASSION system, we are in the process of using PASSION run-time library for I/O in several real parallel applications or their templates. LU factorization, a widely used scientific kernel in the solutions of linear systems, is the application that was examined and implemented both using PASSION and without using PASSION version, in order to show the performance advantageous of PASSION system.

The rest of the paper is organized as follows. Out-of-core computation model in PASSION is explained in

Section 2. Section 3 describes LU factorization. Implementation issues are discussed in section 4. We discuss performance results on Intel Paragon in Section 5, followed by the conclusions.

2 Out-of-core Computation Model in Passion

PASSION (Parallel and Scalable Software for Input-Output) is aimed to supply software support for parallel I/O on distributed memory parallel computers. Compiler, runtime and file system support is provided by the PASSION runtime library. The interface uses collective I/O, which increases I/O efficiency by cooperating the processors. With the PASSION system, user is released from the burden of using explicit operations and tedious works.

Grand challenge applications require large data set that can not fit into memory for the entire duration of a run. These applications are usually implemented in a way that large amount of data resides on secondary storages, which is not in-core but out-of-core; and at any time only a portion of it resides in memory. Out-of-core implementations of applications are not common since the implementations are tedious and the performance is very poor. PASSION supports two placement models for storing and accessing data: Local Placement Model(LPM) and Global Placement Model (GPM). Input-Output of data in these models are user-transparent and it is handled automatically by the PASSION Library [2].

- **Local Placement Model**: Global data array of the application is divided into smaller local arrays. Each local array belongs to a different processor and stored in a separate file called the *Local Array File (LAF)* of that processor. Each processor explicitly reads from or writes into its own Local Array File. LAF of each processor is stored on the logical disk of that processor. At any time only a portion of local array can be stored in main memory. The portion of the local array which is in the main memory is called the *In-Core Local Array (ICLA)*. During a run, parts of the Local Array File are fetched into a ICLA where the in-core computations are performed and ICLA is stored back to the appropriate locations in the Local Array File.

- **Global Placement Model**: In this model the global array is stored in a single file called the *Global Array File*. No local arrays are created as in the Local Placement Model. There is a single global array on the disk.

PASSION runtime system automatically fetches the appropriate portion of each processor's local array from the global array file.

Each model has both advantages and disadvantages onto the other one. The advantage of the Global Placement Model is to save the cost of initial local array creation phase required in the Local Placement Model. Its disadvantage Placement Model is that each processor's data may not be in a continuous manner; which results in a higher I/O latency time. However, this drawback can be overcome to a large extent by using collective I/O. In this paper, Global Replacement Model was chosen as the data access and storage strategy for the given LU factorization application.

3 LU Factorization

LU factorization [5] of a given $n \times n$ matrix is the matrix multiplication of unit lower triangular matrix L and nonunit upper triangular matrix U. If partial pivoting is performed then

$$PA = LU$$

where P is the permutation matrix which represents the accumulation of all pivots required for stability. If the factorization phase is completed, $Ax = b$ linear system can be solved by means of a forward followed by a backward substitutions. phases:

$$Ly = Pb \text{ and } Ux = y$$

Substitution phase is trivial and do not require as much I/O as in the factorization phase, therefore we concentrated on only the LU factorization kernel in this paper.

In-core LU factorization can be implemented using two main methods: *direct LU factorization* [5] and *block LU factorization* [6]. Direct LU factorization algorithm given in Figure 1 modifies all columns to the right after scaling a column with its pivot element. In this algorithm, lower triangular part of matrix A (with unit diagonal) gives the L matrix; and upper triangular part of matrix A gives the U matrix.

```
DO $i = 1$ to $n - 1$
    DO $j = i + 1$ to $n$
        /* Update the pivot column */
        $A(j, i) = A(j, i)/A(i, i)$
    END DO
    DO $j = i + 1$ to $n$
        $k = i + 1$ to $n$
            /* Update the reduced part of the matrix */
            $A(j, k) = A(j, k) - A(i, k).A(k, j)$
        END DO

    END DO
END DO
```

Figure 1: Sequential Version of Direct LU factorization

It is also possible to organize the LU factorization so that matrix operations become the dominant part, as in the *Block LU factorization.* In block LU , matrix is partitioned as follows

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

where A_{11} is a $r \times r$ matrix, A_{12} is a $r \times (n-r)$ matrix, A_{21} is a $(n-r) \times r$ matrix, and A_{22} is a $(n-r) \times (n-r)$ matrix, r is a blocking parameter, and n is the size of each dimension of the original matrix.

Block LU algorithm firstly computes the LU factorization for A_{11}, i.e, $A_{11} = L_{11}U_{11}$. Now we have both L_{11} and U_{11} parts. The next step is to solve the following two triangular systems. Figure 2 shows all steps of the recursive block LU factorization.

Both direct LU and block LU can be used for out-of-core LU factorization. In the out-of-core version, when the columns to the right are not in the memory, their modification is postponed until the slab containing them is read into memory. One approach is to read and update all remaining slabs in every column factorization step. Another alternative is that when the slab is in memory , each processor must read the columns on the left to

```
Compute LU factorization for $A_{11}$, i.e $L_{11}U_{11} = A_{11}$.
Solve triangular system $L_{11}U_{12} = A_{12}$.
    Overwrite $A_{12}$ with $U_{12}$ matrix.
Solve triangular system $L_{21}U_{11} = A_{21}$.
    Overwrite $A_{21}$ with $U_{21}$ matrix.
Update $A_{22} = A_{22} - L_{21}U_{12}$ and recursively compute $A_{22}$.
```

Figure 2: Recursive Version of Block LU factorization

obtain the data which is required to update the postponed operations. In this paper, Direct LU implementation with the former update approach was used in the implementations.

4 Implementation

LU factorization has been implemented on Intel Paragon (L38) machine at Caltech JPL Lab. The Paragon L38 consists of 512 computing nodes, 6 source nodes, 21 MIO nodes and 3 HIPPI network nodes that are configured as a two dimensional mesh with 16 rows and 36 columns. Intel i860 XP microprocessor was used in the computing nodes, which has a peak performance of 75 MFlops. Disk subsystem consists of 21 MIO nodes and a RAID controller.

In the factorization, the matrix entries were distributed to the processors using a row-cyclic decomposition (as shown in Figure 3), in order to fully support load-balancing. I/O distribution is a column-block distribution. Figure 3b shows the distribution of a 16×16 matrix onto 4 processors. ICLA size is 4×4 and therefore total matrix can be read in 4 I/O operations by all processors. In the implementation, in order to decrease the total number of bytes transferred per I/O operation, number of rows in ICLA matrix is decreased by one in every 4 steps; since number of nodes equals to 4. In every 4 steps all nodes factor their current row as shown in Figure 3b.

LU factorization has been implemented both using PASSION runtime library and using conventional parallel I/O calls, in which low-level operations have been implemented explicitly. In latter, direct file access strategy was used. Access pattern of PASSION version was based on collective I/O strategy, where requesting processors

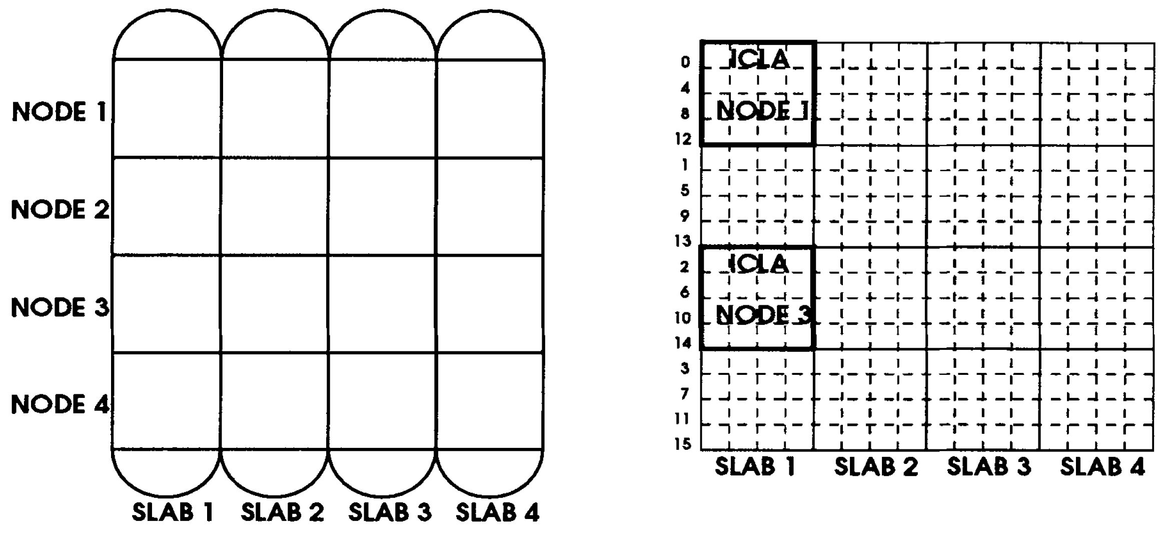

Figure 3: Distribution of a 4x4 matrix on 4 processors

cooperate in reading and writing data in an efficient manner.

Pseudocode for the Paragon implementation is shown in Figure 4. The communication primitives, *csend* (for sending data to one or more nodes) and *crecv* (for receiving data), were used in the following parts of the code:

- Partial Pivoting. In that part, two subroutines are used for the implementation. *Local_Pivot* subroutine is the subroutine for finding the pivot element of each processor; *Global_Pivot* subroutine is a binary-exchange routine which takes order $O(\log P)$ to find out the global pivot element. After the call to *Global_Pivot* routine, every node gets the pivot element and its owner.
- Exchange the pivot row and the current reduced row.
- Broadcast pivot row to all processors. Each slab requires its pivot row portion, in order to update the slab.

Storage type of the input matrix was considered both row-major and column-major. Therefore for each matrix size there are four test cases:

- conventional version with row_major storage
- conventional version with column_major storage
- PASSION version with row_major storage

- PASSION version with column_major storage

Passion_global_read and *Passion_global_write* routines handle the required I/O operations in the application; whereas in the conventional versions these calls were implemented explicitly by *read_from_file* and *write_into_file* routines. When the *read_from_file* routine is called, it sets the seek pointer to the given offset, with the *lseek* command. The offset value is the beginning address of the ICLA in each processor. After the pointer is set to the appropriate offset, ICLA data is read from the file with an synchronous read command, *cread.* Then the data is copied into ICLA with an efficient memory copy command, *memcpy.* In write_to_file routine, seek pointer is set in a similar manner and then ICLA is damped into file with a synchronous write command, *cwrite.*

For the I/O routines in conventional version using row-major storage, seek pointer is set in every row of ICLA; i.e, for a 4×4 ICLA there are 4 lseeks in both read_from_file and write_to_file routines. However in column-major order there are two options: either seek pointer will be set for each item in the ICLA and then they will be copied one by one; or seek pointer will be set for each column as in the row-major case, but they will be moved to their places in ICLA, one-by-one. Since it is clear to see that both methods of conventional version for column-major storage will give worse results than row-major storage, only conventional version with row-major storage was used as part of the test case.

```
DO j=1 to n-1
    /* Update column j */
    Calculate the current slab number (in which column j exists)
    Get the slab containing column $j$ from GAF into ICLA. (CALL Passion_Global_Read).
    /* Local pivot calculation */
    $pivot_m = \max |ICLA(i,j)|$ where $i : begin\_ICLA\_col \rightarrow end\_ICLA\_col$
    /* Global pivot calculation in log $P$ steps using binary exchange approach */
    $pivot_{global} = \max pivot_i$ where $i : 0 \rightarrow no\_of\_nodes - 1$
    IF ($mynode$ is the owner of the pivot element) THEN
        BROADCAST the pivot element to the other nodes
    ELSE
        RECEIVE pivot element from its owner.
    DO current_slab= init_slab to no_of_slabs
        /* Update all slabs of the node $m$ (for column $j$) */
        Get the current_slab from GAF into ICLA. (CALL Passion_Global_Read).
        IF (current_slab is the initial_slab) THEN
            /* Update pivot column (in all nodes) and store results in a vector */
              $ICLA(k,j) = ICLA(k,j)/pivot_{global}$
              $update_{column}(k) = ICLA(k,j)$
        IF ($mynode$ is the owner of the pivot element) THEN
            BROADCAST the $pivot_{row}$ to the other nodes
        ELSE
            RECEIVE receive $pivot_{row}$ and store it in a vector called $update_{row}$ .
        /* Update ICLA and restore it into GAF */
        DO k=begin_ICLA_row to end_ICLA_row
            DO l=begin_ICLA_col to end_ICLA_col
              $ICLA(k,l) = ICLA(k,l) - update_{row}(k) * update_{col}(l)$
            END DO
        END DO
        Write the current_slab from ICLA into GAF. (CALL Passion_Global_Write).
    END DO
END DO
```

Figure 4: Node Program for LU factorization

5 Results

We implemented the algorithms described above on Intel Paragon at Caltech, using C programming language with Paragon's communication primitives. PASSION run-time library was loaded with the help of compiler switches. In the experiments, matrix size varies from 128×128 to 512×512. For each input matrix, ICLA size is changed from 16×16 to 64×64. In each test case, timings were recorded after 10 runs, deleting the best and the worst ones and averaging the remaining.

Figure 5 shows how Collective I/O is implemented in PASSION. Data matrix is distributed on three processors. In collective I/O implementation, bounding box, i.e., current slab the of the matrix which will be read by processors, is distributed equally among processors. Processors will read their assigned parts and then they will collectively exchange data in order to get their actual part. Figure 5a is for column-major storage where the first bounding box exactly fits the first slab from the file; therefore all read data is used, none is wasted. However in row-major storage collective I/O reads more unused data than used ones, in each step; which means PASSION LU implementation for row-major storage will be worse than column major storage (Figure 7 and Figure 8). The shaded regions in Figure 5 show the first part which will be read by each processor.

Figure 7, and Figure 8 compares the I/O performance of Passion (for both row and column-major storage) and Conventional implementation (for row major only) of LU algorithm on 128×128 and 256×256 matrices, respectively. Row-major alternative of conventional version was not used as a comparison case because of the reasons explained in page 8. We observe that LU factorization using PASSION runtime library (for a column-major stored file) performs much better than the conventional implementation. The difference becomes larger if number of processors used is increased; since Collective I/O (as in PASSION version) gives better results for large processor grid than a small one.

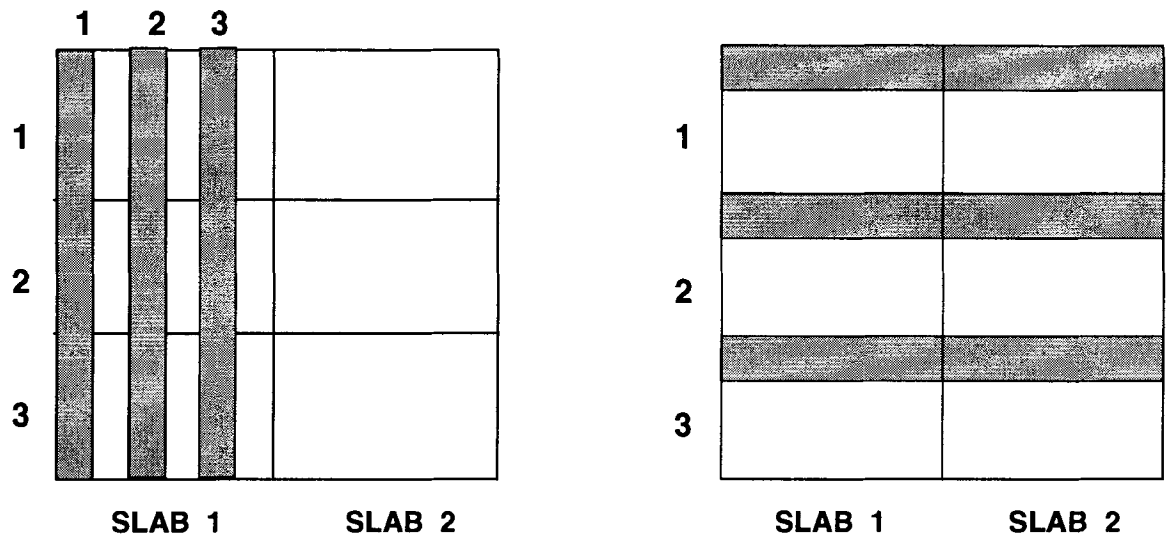

Figure 5: Collective I/O portions for column-major storage on the left, row-major storage on the right.

Matrix	ICLA	Nodes	I/O	Conventional V.		PASSION (row)		PASSION (column)	
Size	Size	Used	Operations	I/O Time	Total Time	I/O Time	Total Time	I/O Time	Total Time
128x128	16x16	8	1150	2462.12	2516.45	3573.75	3588.98	743.15	763.16
128x128	32x32	4	638	1294.02	1314.74	1568.43	1574.60	794.9	800.51
128x128	64x64	2	382	724.64	731.21	1398.82	1406.4	891.4	896.96
256x256	16x16	16	4350	20670.01	21181.76	30011.11	30118.45	2801.09	2869.38
256x256	32x32	8	2302	10693.57	10812.22	14873.35	14854.12	3054.54	3094.05
256x256	64x64	4	1278	5579.87	5624.84	7882.23	7898.50	3423.97	3440.06

Figure 6: Timing of the LU Factorization

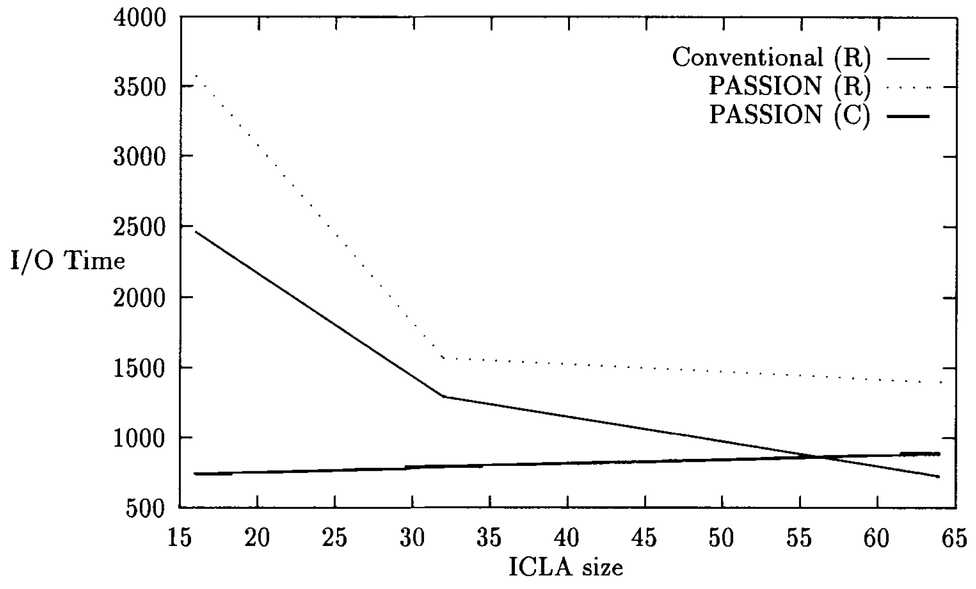

Figure 7: I/O Timing for 128 × 128 matrix size

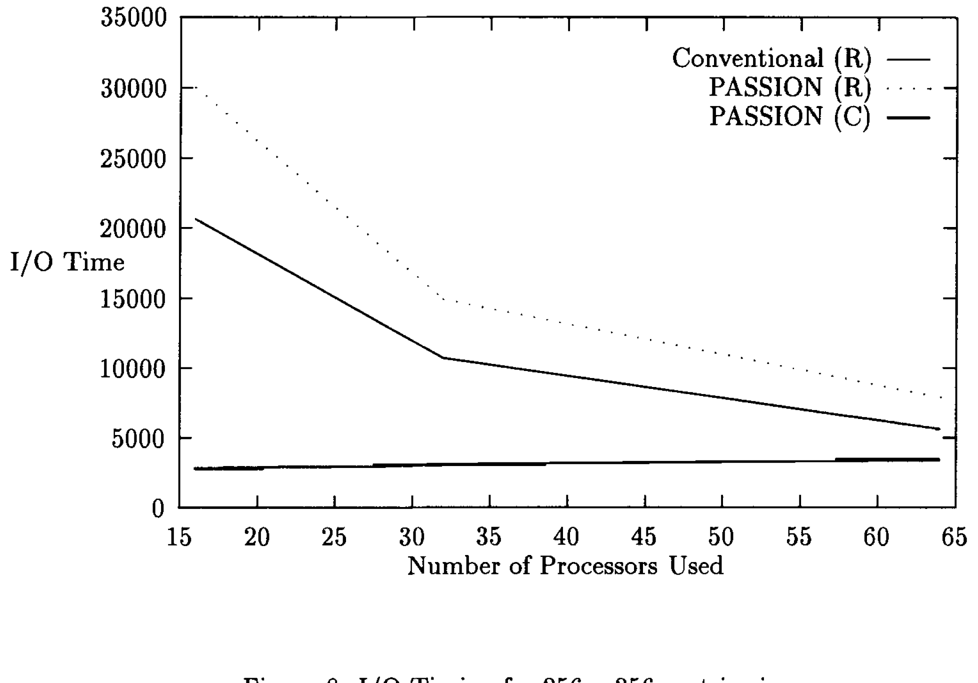

Figure 8: I/O Timing for 256 × 256 matrix size

References

[1] del Rosario M., Choudhary A., High-Performance I/O for Massively Parallel Computers, *IEEE Computer*, March 1994.

[2] Choudhary A., Bordawekar R., Harry M., Krishnayer R., Ponnusamy R., Singh T., and Thakur R., PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS-636, NPAC, Syracuse University, September 1994.

[3] Bordawekar R., Choudhary A., Kennnedy K., Koelbel., Palenczny M., A Model and Compilation Strategy for Out-of-Core Data Parallel Programs.

[4] Thakur R., Bordawekar R., Choudhary A., Compiler and Runtime Support for Out-of-Core HPF Programs, *Proc. of* 8th *ACM International Conference on Supercomputing* , Manchaster, England, July 1994.

[5] Golub G., Loan C., Matrix Computations, Second Edition, The John Hopkins University Press, Baltimore and London, 1993.

[6] Womble D., Greenberg D., Riesen R., Wheat S., Out of Core, Out of Mind: Practical Parallel I/O.