Load balancing in pre-processing of large-scale distributed sparse computation

Olfa HAMDI-LARBI
Faculty of Sciences of Tunis, CS Department
and University of Versailles, PRiSM Laboratory
ola_ola79@yahoo.fr

Zaher MAHJOUB
Faculty of Sciences of Tunis, CS Department
Belvedere 1060 Tunis TUNISIA
Zaher.mahjoub@fst.rnu.tn

Nahid EMAD
University of Versailles, PRiSM Laboratory
78035 Versailles Cedex FRANCE
Nahid.emad@prism.uvsq.fr

Abstract In this paper, we study the sparse matrix-vector product (SMVP) distribution on a large scale distributed system (LSDS). The framework is defined by three steps: pre-processing, processing and post processing. We focus here only on the first step i.e. ***pre-processing***. Our general goal is to detect, for a given sparse matrix, the best compression format i.e. which leads to the best performances for the SMVP on the LSDS. Thus, we need to study different approaches to distribute the data, essentially the sparse matrix. Three approaches are proposed. In the first one, the matrix A is partitioned into row blocks with the same number of rows. The second (resp. third) approach consists in partitioning A into blocks of contiguous (resp. non contiguous) rows with the same number of non-zero elements. These approaches result in solving some constrained scheduling problems of the NP-hard type. To solve them, we propose some multi-phase heuristics. After presenting these approaches, we present for each one a performance analysis. These analyses are then validated through a series of experimentations that permit to show their interest.

Keywords: sparse computation, distributed computation, pre-processing, load balancing.

1. Introduction

In numerical scientific computing, many problems reduce to large sparse linear system resolution and/or eigen-elements computation. The iterative methods allowing to solve these problems are based onto a sequence of sparse matrix-vector products (SMVP) with the same matrix but different vectors. Our aim here is to study the SMVP distribution on a LSDS, particularly sparse matrix distribution and load balancing. In principle, three steps would be necessary to achieve this kind of applications. The first one is pre-processing which generally reduces to data preparation, code optimisation [EHM05], performance prediction, etc. The second step, namely processing, corresponds to computations and communications done by the application. As to the third step i.e. post processing, it consists in analysing the application results. In this paper, we are focused only by the first step i.e. pre-processing. Our aim is to detect, for a given sparse matrix, the best compression format that provides the best performances for the SMVP on a large-scale distributed environment. Thus, we need to study different approaches to distribute the data concerned by the processing step. Indeed, load balancing at the level of processing units could have a dramatic influence on application performances.

Sparse matrix distribution is a particular scheduling problem thus known to be a hard problem. Therefore, we thought of approximation algorithms to solve it and designed (three) different approaches heuristics based. They both use one dimension of the matrix (row or column). In this paper, we restrict to row fragmentation (RF) as column fragmentation (CF) is exactly the symmetric of RF. The choice of 1-D fragmentation, and especially of row dimension, is justified by the fact that our target environment is a large scale distributed system (LSDS) e.g computation grid, Peer-to-Peer system, etc. As a matter of fact, a 2-D [BiM05][VaB05] or a column fragmentation generates supplementary communications (compared to RF) that are needed to achieve the SMVP resulting vector reduction. However, in an LSDS, communications are very expensive since they are generally done via Internet. In the pre-processing study, we take into account neither system heterogeneity nor nodes volatility. The last are taken into account at the application code level i.e. at processing level which will be the topic of a forthcoming paper.

The remainder of the paper is organised as follows. In section 2, we present an overview of sparse matrices, sparse computations and Peer-to-Peer systems. In section 3, we describe three approaches (heuristics) for sparse matrix row fragmentation. Section 4 is devoted to complexity and heuristics analyses. In section 5, we expose and comment the results of the fragmentations we applied on randomly generated matrices as well as on banded matrices. Finally, in section 6, we present a conclusion and perspectives.

2. Context and definitions

2.1 Sparse matrices

Let A be a (real) N×N large matrix. If NNZ, the number of non-zero elements of A is very few (resp. large), let us say NNZ=O(N) (resp $O(N^2)$), then A is called *sparse* (resp. *dense*). Both storage requirements and computational time of any application operating on large sparse matrices (SM) may be dramatically reduced by only storing non-zero elements and avoiding redundant operations on zero elements [Bik96]. This is achieved by using compressed storage formats (CSF) for SM's. A sparse matrix may have various structures according to the locations of its non-zero elements. An SM structure (SMS) may be either regular e.g. triangular, diagonal, constant-banded, Hessenberg, etc; or irregular (also called general) e.g. variable-banded, random, etc [GoV83][DER92][Saa94]. For each SMS, one or more dedicated CSF's are known in the literature. For instance, to a general structure we can associate CSR (Compressed Storage Row) or CSC (Compressed Storage Column) ; for a (regular) triangular structure we can find MSR (Modified Storage Row) and MSC (Modified Storage Column) ; for a banded one we have BND (BaNDed), DIA (DIAgonal), JAD (JAgged Diagonal), etc [Saa94]. In this paper, we are essentially interested in CSR (Compressed Storage Row) format. This is due to the fact that it is the most used in practice.

Assume that our N×N matrix A is stored in a two dimensional array (non compressed format). Storing A in the CSR format consists in using three 1-D arrays (vectors) denoted A, JA and IA defined as follows [Saa94][Jen80][GoV83] :

- A is a real array involving the non-zero elements values a_{ij} (of matrix A) stored row-wise (from row 1 to row N). A is of NNZ, where NNZ is the number of non-zero elements of A.
- JA is an integer array involving the column indices of the stored elements in A. JA is of length NNZ.
- IA is an integer array involving the pointers on the head of each row in arrays A and JA. Thus, IA(i) is the position in A and JA where begins the i-th row of A.

We have done some statistics on selected matrices from Matrix Market [Mat06] and we have remarked that non zero elements distribution in a sparse matrix rows is even non uniform. We will essentially take into account this kind of matrices witch seems to represent the most frequent cases coming from real applications.

2.2 Sparse computation

As mentioned above, sparse computation refers to algorithms processing SM's [CLS98]. In fact, several large scale applications in scientific computing reduce to the resolution of sparse matrix problems e.g. in molecular dynamics, fluids dynamics, signal/image processing, etc [Ase96]. Particularly, various sparse scientific computations reduce to the resolution of sparse linear algebra problems. In fact, the two main problems encountered in this last field are the linear system resolution (LSR) [DER92][Jen80] and the eigenvalues computation (EVC).

Contrary to the eigenproblem for which we have to use an iterative method, the techniques allowing to solve a linear system can be iterative [GoV83] or direct [DER92]. Direct methods are robust and generally used for problems with moderate size. Otherwise, iterative methods are unavoidable. Moreover, for our targeted sparse large-scale problems, direct methods complexities can be prohibitive. As a matter of fact, they are too memory and time consuming because of the *fill-in* phenomenon. Therefore, for this class of problems, iterative methods are an interesting alternative [DER92]. The sparse matrix-vector product, where the matrix is sparse and the vector is dense, constitutes, in fact, a highly used kernel in iterative methods for sparse LSR and EVC problems.

2.3 Peer-to-Peer system

Often simply referred as P2P, peer-to-peer architecture is a wide-area network of work stations (peers). The workstations in the system have the same responsibilities (each of them can be client, server, worker, etc) and can be heterogeneous and volatile. This differs from client/server architectures where some computers are dedicated to serving the others [Web05]. In this paper, we propose several approaches to optimise the pre-processing of the sparse matrix distribution on a large-scale platform such as a P2P system. Thus, our proposed solutions are based on 1-dimension fragmentation, and especially of row dimension.

This is because, a 2-dimension [BiM05][VaB05] or a column fragmentation generates supplementary communications (compared to row fragmentation) on these targeted environments. This is due to the reduction of the SMVP resulting vector which can be very expensive since it is generally done on Internet. In the pre-processing study, we take into account neither system heterogeneity nor nodes volatility. The last are taken into account at the application code level i.e. at processing level which will be the topic of a forthcoming paper.

3. Data fragmentation

In the following, we present three alternatives for decomposing a SM into fragments (blocks) whose number is denoted *NbFrag*. The first one consists in decomposing the matrix, denoted A, into row blocks (fragments) of same size i.e. involving the same number of rows. Two different algorithms are proposed for this approach, FSNR (Fragmentation with Same Number of Rows) and GFSNR (Generalised FSNR). The second, called FSNZ (Fragmentation with same Number of Non-zeros) consists in decomposing A into fragments of contiguous rows (not necessarily having the same number of rows) involving, as much as possible, the same number of non-zero elements. As to the third approach, it consists in decomposing A into fragments with non contiguous rows and involving the same number of non-zero elements. Two algorithms have been designed here i.e. GFSNZ (Generalised FSNZ) and S_GFSNZ (Sorted GFSNZ).

3.1 Approach 1 : FSNR and GFSNR decompositions

Let q and p be two integers such that N = q * *NbFrag*+ p. The principle of this approach consists in constructing *NbFrag* fragments involving the same number of contiguous rows. Two algorithms are proposed namely FSNR and GFSNR.

FSNR decomposition :
It is a naïve one-phase algorithm that consists in partitioning A into *NbFrag* fragments (row blocks) such that the first p ones involve each q+1 contiguous rows whereas the remaining (*NbFrag*-p) ones involve q.

GFSNR decomposition :
It is a two-phase algorithm (Generalized FSNR) that improves FSNR according to a particular criterion (see below). Remark that in FSNR, the fragments involving q+1 (resp. q) rows are successive i.e. fragment #1… fragment #p (resp. fragment #p+1… fragment #*NbFrag*). In phase 1 of GFSNR, the p (resp. *NbFrag*-p+1) fragments involving q+1 (resp. q) rows are randomly chosen (i.e. are not necessarily successive).
Concerning phase 2 which represents an iterative heuristic, let us first define the criterion, denoted CRIT, we aim to optimize. It could be either the maximal load (ML: the number of non-zeros in the most loaded fragment) or the imbalance factor (IMF) which may be either the difference between the maximal and the minimal loads (AIMF: Absolute IMF) or the ratio of the two latter (RIMF for Relative IMF).
As far as we are concerned, we have chosen the AIMF criterion. The improving procedure consists in reducing the maximal load and increasing the minimal one by row interchange between successive fragments. This permits, through successive iterations, to refine the former fragmentation, hence leading to a better balanced one.
To be more precise, let *maxfrag* be the maximally loaded fragment. Its first (or last) row is picked and assigned to its preceding (or succeeding) fragment, provided that this does not increase the current maximal load.
In a symmetric way, let *minfrag* be the minimally loaded fragment. The first row of its succeeding fragment (or the last row of its preceding one) is picked and assigned to it, provided that this does not decrease the current minimal load. This procedure is iterated as long as the criterion may be reduced or a fixed number of iterations is not exceeded.

3.1.2 Approach 2: FSNZ decomposition

The approach for data decomposition presented above is quite naive. It has to be underlined that when the non-zero elements are non uniformly distributed in the matrix rows, FSNR and GFSNR may can induce a prohibitive load imbalance for peers.
The FSNZ decomposition has been designed in order to avoid such drawback. It consists in decomposing matrix A into row blocks in such away that each block (fragment) involves (as much as possible) the same number of

non-zero elements. Remark that the number of non-zero elements per fragment should be around NNZ/*NbFrag* in order to guarantee a fair decomposition.
Furthermore, notice here that (i) a row does belong to only one fragment, (ii) the number of rows is not necessarily the same for all the fragments and (iii) each fragment is constituted by contiguous rows. Hence, each fragment has to be identified by its first row index.
It has to be underlined that the construction of the balanced FSNZ decomposition, is in fact a particular (*constrained*) scheduling problem (SP) [Hoc97] [SaY99] of N (non pre-emptive) independent tasks (i.e. the rows of A) with corresponding costs c_1, c_2, ...,c_N (c_i is the number of non-zero elements in row i) onto *NbFrag* homogeneous processors under the constraint that the tasks (i.e. rows) assigned to any processor (i.e. fragment) must be successive (i.e. contiguous).
To solve this particular SP, an NP-hard one [Hoc97], several polynomial time approximate algorithms (heuristics) are known in the literature [Hoc97] [SaY99]. As far as we are concerned, we designed a two-phase heuristic.
The first phase, a constructive heuristic, permits to construct an initial fragmentation in O(N) time. It consists in assigning contiguous rows to a given fragment until its load (i.e. the total number of non-zeros in its rows) nearly reaches or exceeds *for the first time* the threshold $\lceil NNZ/NbFrag \rceil$. A dynamic computing of the load of the not yet assigned rows (remaining sum of the non-zero elements) permits to (i) decide either to exceed or not $\lceil NNZ/NbFrag \rceil$, (ii) guarantee that any fragment involves at least one row and (iii) avoid too large load imbalance.
Concerning Phase 2, it is similar to Phase 2 of GFSNR algorithm with the only difference that for FSNNZ we eliminate the constraint concerning the number of rows in each fragment which is either q or q+1 in GFSNR.

3.3 Approach 3: GFSNZ and S_GFSNZ decompositions

In FSNZ, we have built fragments under the constraint that rows involved by any fragment are necessarily contiguous in matrix A. As the non-zeros distribution in A is random, this decomposition doesn't guarantee an optimal load balancing among the fragments. An alternative consists in relaxing the above constraint i.e. a fragment may involve non contiguous rows. Here, we have to solve a new problem which is the same scheduling problem (SP) seen above but with no constraint. For this purpose we designed two heuristics GFSNZ (Generalised FSNZ) and S_GFSNZ (Sorted GFSNZ).

S_GFSNZ decomposition
It is a three-phase heuristic. The first one (phase 0) consists in sorting the rows of A in increase order of their number of non-zero elements. This may be done in $O(N*\log_2 N)$ time. As the second phase, we make use of a classical (constructive) heuristic [SaY99] that consists first in assigning row i to fragment i (i=1…*NbFrag*). Then, the next row is assigned to the less loaded fragment and so on. In other words, the current non assigned row is always assigned to the less loaded fragment until row emptying.
The third phase, the improving one, is an iterative heuristic. It permits, through successive iterations, to refine the former fragmentation, hence leading to a better balanced one. Choosing the AIMF criterion, the designed procedure consists in successive row interchange between the maximally and minimally loaded fragments.
More precisely, let *maxfrag* (resp. *minfrag*) be the maximally (resp. minimally) loaded fragment. An appropriate row from *maxfrag* and an appropriate row from *minfrag* are interchanged so that AIMF is reduced (at the best). This procedure is iterated as long as the AIMF criterion may be reduced or a fixed number of iterations is not exceeded.

GFSNR decomposition
We could avoid phase 0 of S_GFSNZ i.e. apply phase 1 without sorting the rows and then phase 2.

4. The data fragmentation algorithms: a priori performance analysis

It deals here with theoretical performance evaluation of the algorithms presented above. We will try to validate these estimations in section 5.

4.1 Approach 1

Clearly, algorithm FSNR can be achieved in O(*NbFrag*) time. Concerning GFSNR, its first phase costs O(*NbFrag*) time and it is easy to prove that its second phase may be done in (at most) O(N*nit) time where nit is the (maximal) number of iterations. Therefore GFSNR is done in O(N*nit) time. If we choose nit=O(1) (resp. O(N)) we'll reach an O(N) (resp. $O(N^2)$) complexity. Concerning GFSNR heuristic, which is greedy oriented, it guarantees an approximation with a ratio not exceeding 2. On the other hand, we may prove that, at the worst case, the value of RIMF is bounded by 3N/2. Indeed, consider the Euclidian division N = *NbFrag**q+p. The worst case occurs when the maximally loaded fragment (*maxfrag*) involves q+1 full rows i.e. its load (q+1)*N and the minimally loaded fragment (*minfrag*) involves q rows where each row involves one non-zero element. So its load is q and RIMF is bounded by $N(q+1)/q=N+N/q \leq 3N/2$ as $q \geq 2$.
On the other hand, as phase 1 of GFSNR is done such that:

- *minfrag* is located between two fragments involving each q rows
- *maxfrag* is located between two fragments involving each q+1 rows

Therefore no improvement could be obtained by phase 2. In this case, the AIMF value is (q+1)*N-q.Remark that in the best case, we obtain an RIMF equal to 1 e.g. when p = 0 and the non-zero elements are uniformly distributed on the matrix rows.

4.2 Approach 2

Phase 1 of FSNZ may is done in O(N) time and Phase 2 in (at most) O(N*nit) time where nit is the maximal number of iterations. Consequently, FSNZ may be done in O(N*nit) time. If we chose nit= O(1) (resp. O(N)), we reach a complexity of O(N) (resp. $O(N^2)$).
Concerning the heuristic quality, at the worst case the RIMF value is O(1). Consider the Euclidian division NNZ = *NbFrag** r + s. Let us assume that *NbFrag*<<N. At the worst case we have the following: (i) *minfrag* involves r-N+2 non-zero elements and *maxfrag* involves r non-zero elements. Indeed, *minfrag* is a fragment non allowed (for a balance purpose) to exceed r even though it needs N-1 elements to reach r+1 ; (ii) the row following the last row of *minfrag* is full (involves N elements) ; (iii) *maxfrag* involves r elements before exceeding r+1 ; (iv) we assume that it is allowed to exceed r+1 and that the row following the last row of *maxfrag* is full. Thus, we obtain a *minfrag* with r-N+2 non-zeros and a *maxfrag* with r+N non-zeros. In this case, we have RIMF = (r+N)/(r-N+2).
The best case occurs, for example, when the rows involve the same number of non-zeros.
Let us add as it was the case for GFSNR, FSNZ is also greedy oriented and guaranties an approximation with a ratio not exceeding 2.

4.3 Approach 3

As we have seen above, S_GFSNZ is a three-phase algorithm. Phase 0 (sorting) can be done in $O(N*\log_2 N)$ time. Phase 1 is a constructive phase which can be done in O(N) time. Concerning Phase 2, it can be done in (at most) O(N*nit) time where nit is the maximal number of iterations. Consequently, S_GFSNZ is done in $O(N*(\log_2 N+nit))$. If we chose nit=O(1) (resp. O(N)), we obtain a complexity of $O(N*\log_2 N)$ (resp. $O(N^2)$.
Consequently, GFSNZ may be achieved in O(N*nit) time that is O(N) (resp. $O(N^2)$) time if nit=O(1) (resp. O(N)).
Concerning the quality of S_GFSNZ and GFSNZ heuristics, at the worst case, we obtain an RIMF equal to N. This occurs, for example, when *NbFrag* = N (so each fragment involves one row) and A involves one row with one element and another with N elements i.e. a full row.
Remark that both GFSNZ and S_GFSNZ are greedy oriented and guarantee approximations with a ratio not exceeding 2.

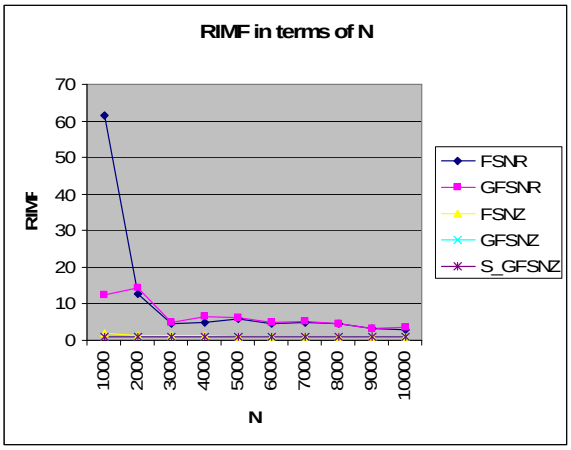
5. Experiments and results interpretation

In order to evaluate the performances of the different fragmentation approaches, we have achieved a series of experimentations on two sets of randomly generated matrices: matrices with general structure and banded matrices.

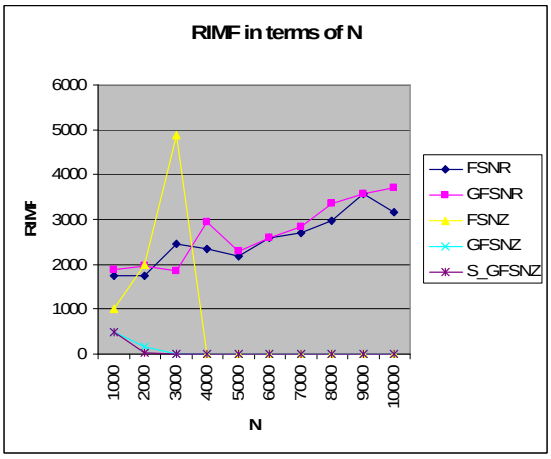
Hence, we compare the different *sequential* algorithms of fragmentation through their RIMF (Ratio IMbalance Factor) variations in terms of matrix size, non zero elements density, number of fragments and bandwidth for banded matrices. The experiments have been done on a Pentium M PC.

5.1 Random matrices

Giving a size N and a density d (=NNZ/N^2), our generator provides a matrix with a random number of non-zeros for each row. In the following, for each fragmentation algorithm, we present curves representing RIMF variations (i) in terms of N, and (ii) in terms in *NbFrag*.

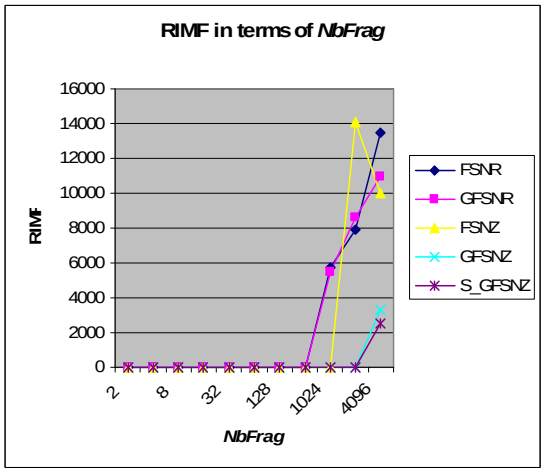


(a) *NbFrag* = 64

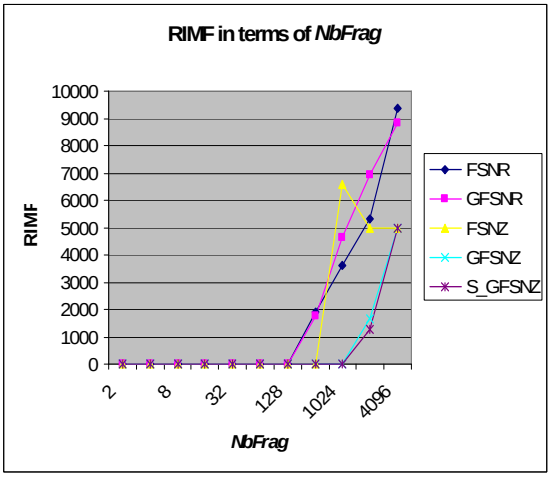


(b) *NbFrag* = 512

Figure 1. RIMF variations in terms of N (density=13%)



(a) N = 10^4



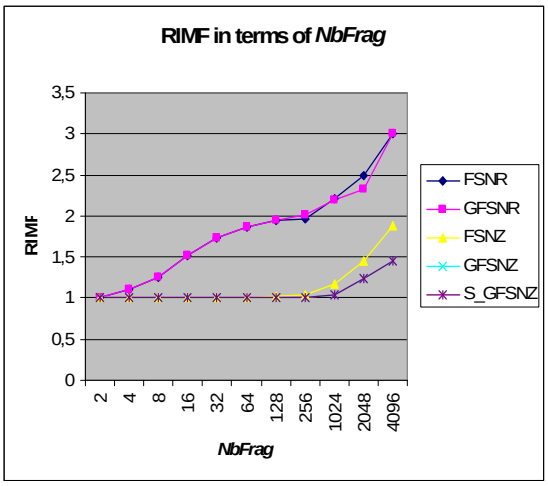
(b) N = 5000

Figure 2. RIMF variations in terms of *NbFrag* (density=13%)

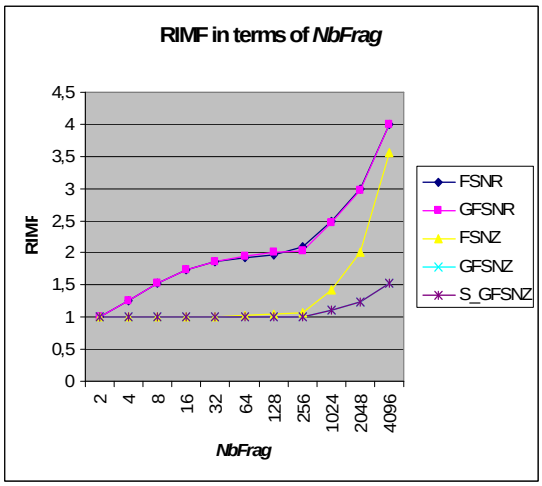
The behaviour of the fragmentation algorithms in term of density d seems to be random. Indeed, the non-zeros distribution has a direct impact on both maximal and minimal loads; the impact of NNZ seems to be null.
Therefore, we remark that the load balancing depends essentially on the number of fragments (*NbFrag*). The larger is *NbFrag*, the larger is the more important is the imbalance. More precisely, it is the ratio N/*NbFrag* which would have a direct impact. In general, we notice that the load imbalance increases when this ratio decreases. This remains true for each fragmentation algorithm. On the other hand, our experiments point out that the algorithms GFSNZ and S_GFSNZ always give the best load balancing and that the ratio RIMF is nearly the same for both. Hence, GFSNZ is more interesting since S_GFSNZ is more expensive because of the initial sorting.
The behaviour of the fragmentation algorithms in term of density d seems to be random. Indeed, the non-zeros distribution has a direct impact on both maximal and minimal loads; the impact of NNZ seems to be null.

5.2 Banded matrices

Giving a size N and a half-bandwidth hbw, the generator provides a banded matrix. In the following, for each fragmentation algorithm, we present curves representing RIMF variations in terms of (i) N, (ii) hbw and (iii) *NbFrag*.

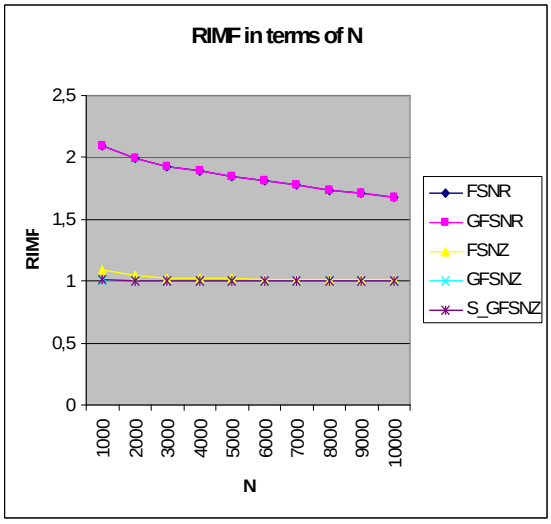


(a) $N=10^4$

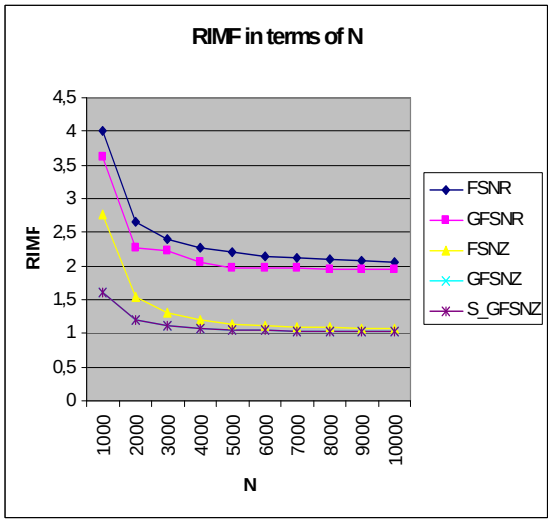


(b) N=5000

Figure 3. RIMF variations in terms of *NbFrag* for hbw=1000



(a) *NbFrag* = 64



(b) *NbFrag* = 512

Figure 4. RIMF variations in terms of N for hbw=400

As seen for random matrices, the number of fragments has an important impact on load balancing. All the fragmentation algorithms have the same behaviour when the matrix size N varies i.e. RIMF decreases when N increases and more precisely when it gets away from the *NbFrag* value. RIMF becomes almost constant when N/*NbFrag* is very low. Notice that GFSNZ and S_GFSNZ always give the best load balance (nearly the same curves). We have determined RIMF variations in terms of hbw and we have remarked that the half bandwidth has not a real impact on lad balancing.

6. Conclusion and further work

In this paper, we have proposed five different approximation algorithms for sparse matrix decomposition: FSNR, GFSNR, FSNZ, GFSNZ and S_GFSNZ. They both consist in partitioning the matrix into row blocks (fragments). We can classify these algorithms into two groups G1 and G2. G1 involves the first three and G2 involves the two others. The algorithms of G1 (resp. G2) produce blocks involving contiguous (resp. contiguous)

rows. Our aim was to obtain balanced fragments i.e. having (nearly) the same number of non-zeros. The designed algorithms, having different time complexities lead to different load balancings. As a matter of fact, in G1 we have an O(*NbFrag*) (where *NbFrag* is the number of fragments) complexity for FSNR and $O(N^2)$ (in the worst case, N being the size of matrix A) for both GFSNR and FSNZ. Each algorithm of the second group is of $O(N^2)$ worst case complexity. Concerning the load balancing, it was measured by using the so called RIMF ratio (maximal load divided by minimal load). Both algorithms of the second group gave equivalent RIMF's. In most cases, the RIMF's in the G2 are better (i.e. lower) than in G1.

The experimental results show that the RIMF value increases with *NbFrag*. This fact occured for random matrices as well as for banded matrices. Notice that for random matrices, when using GFSNZ algorithm (group G1), RIMF gets farther from value 1 when *NbFrag* exceeds 256 (we considered matrices of size 5000 and 10^4). The same situation happens with banded matrices for *NbFrag* larger than 1024. Thus, we have better reducing the number of blocks when fragmenting the sparse matrix.

We also notice that for random (resp. banded) matrices, the density (resp. half bandwidth) has no impact on RIMF value. Therefore, the choice of an adequate (moderate) number of fragments is the key for a good load balancing. Our current work, by taking into account the results presented in this paper, consists on the processing and post-processing of SMVP on large-scale peer to peer systems. Thus, we intend in the near future to experiment the different fragmentation algorithms for the (SMVP) kernel on a real Peer-to-Peer (P2P) system**.**

7. References

[Ase96] R. ASENJO & al., On the Automatic Parallelization of Sparse and Irregular Fortran Codes, Technical Report N°UMA-DAC-96/34, University of Illinois, 1996.

[Bik96] A. J. C. BIK, Compiler Support for Sparse Matrix Computations, PhD Thesis, University of Leiden, Netherlands, 1996.

[BiM05] R. H. BISSELING & W. MEESEN, Communication balancing in parallel sparse matrix-vector multiplication Electronic Transactions on Numerical Analysis, Special Issue on Combinatorial Scientific Computing, 21, pp. 47-65, 2005.

[CLS98] B. L. CHAMBERLAIN, E. C. LEWIS, L. SNYDER, A Region-based Approach for Sparse Parallel Computing, UW CSE Tech. Rep. UW-CSE-98-11-01, 1998.

[DER92] I. S. DUFF, A. M. ERISMAN & J. K. REID, Direct Methods for Sparse Matrices, Oxford Science Publications, 1992.

[EHM05] N. EMAD, O. HAMDI-LARBI & Z. MAHJOUB, On sparse matrix-vector optimization, ACS / IEEE International Conference on Computer Systems and Applications (AICCSA '05), Cairo, 2005.

[GoV83] G. H. GOLUB, C. F. VAN LOAN, Matrix Computations, Johns Hopkins University Press, 1983.

[Hoc97] D.S. HOCHBAUM (ed), A*pproximation algorithms for NP-Hard problems*, PWS Publ. Co., 1997.

[Jen80] A. JENNINGS, Matrix Computation for Engineers and Scientists, John Wiley & Sons, 1980.

[Mat06] http://math.nist.gov/MatrixMarket/, 2006.

[Saa94] Y. SAAD, SPARSKIT: a Basic Tool Kit for Sparse Matrix Computation, http://www.cs.umn.edu/Research/arpa/SPARSKIT/paper.ps , 1994.

[SaY99] S.M. SAIT & H. YOUSSEF, *Iterative computer algorithms with applications in engineering*, IEEE C.S ed., 1999.

[VaB05] B. VASTENHOUW & R.H. BISSELING, A two dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Review, Vol. 47, No. 1, pp. 67-95, 2005.

[Web05] http://www.webopedia.com, 2005.