Performance Evaluation of Partitioning Algorithms for Parallel Sparse Computations

Sana Ezouaoui [1], Olfa Hamdi-Larbi[1], ZaherMahjoub[1]
[1]University of Tunis El Manar, Faculty of Sciences of Tunis, URAPOP, 2092, Tunis, Tunisia
sana.ezouaoui@fst.utm.tn ; olfa.hamdi@utm.tn ; zaher.mahjoub@fst.rnu.tn

***Abstract*—In this paper, we are interested in sparse matrix partitioning for parallel computation. As a matter of fact, sparse computing such as Sparse Matrix Product (SMP i.e. product of two matrices where at least one is sparse) and Sparse Matrix-Vector Product (SMVP) are kernels widely used in real world scientific applications. Our general goal is to study the parallelization, in a shared memory environment, of the SMP where only the first matrix is sparse as well as the SMVP by designing suitable partitionings of the sparse matrix into blocks (fragments) of either contiguous or non contiguous rows in order to ensure fair inter-processor load balancings. Contiguous partitioning (CP) is in fact an easy combinatorial optimization problem (COP) that may be solved by exact polynomial algorithms (e.g. Dynamic Programming algorithms) or approximation algorithms (e.g. greedy heuristics) of linear complexity. On the other hand, non-contiguous partitioning (NCP) is a NP-hard COP for which several polynomial heuristics may be used. We propose here heuristics involving different versions for solving both CP and NCP cases. Besides a theoretical analysis of the different heuristics, we carried out a series of experiments where the input data are benchmark and randomly generated matrices. This permitted to establish a performance evaluation of the designed heuristics and accurate comparisons between them, thus underlining the practical interest of the study.**

Keywords—combinatorial optimization problem, contiguous partitioning, heuristics, non-contiguous partitioning,load balancing, parallel algorithm , parallel sparse computing, scheduling, sparse matrix.

I. INTRODUCTION

Sparse computing is an understudied primitive in numerical linear algebra [1], [2]. SMP is already a common operation in computational linear algebra, usually and repeatedly used within the context of block iterative methods [2].

Sparse matrix kernels constitute the computational basis of many scientific and engineering applications [2]. SMP has applications in diverse domains such as the all-pairs shortest-paths problem in graph analytics, non-negative matrix factorization for dimensionality reduction, a novel formulation of the restriction operation in Algebraic Multigrid, quantum Monte Carlo simulations for large chemical systems, interior-point methods for semi-definite programming, the siting problem in terrain modeling, sparse inverse covariance selection used for data analysis, etc [1],[3]–[5].

Parallelization of SMP requires the decomposition and distribution of the sparse matrix. Two objectives in the decomposition are the minimization of communication requirement and load imbalance. For this purpose, we use either graph and hypergraph partitioning models [2], [6] or scheduling list model optimisations [7]. The hypergraph model is often used when targeting a distributed memory architecture since it minimizes the total communication overhead volume [2], [4]. As we target here a shared memory architecture, we are interested in scheduling list model optimisations. Data partitioning are in fact categorized into 1D, 2D, and 3D partitioning [8]. In this paper, we specially focus on 1D decomposition since we use GAXPY_R body kernel [5], [9], [10].

Sparse matrix distribution is a particular scheduling problem where the sparse matrix may be partitioned into blocks, also called fragments, of contiguous or non-contiguous rows or columns. In this paper, we restrict to row fragmentation (RF) as column fragmentation (CF) is exactly the symmetric of RF. Contiguous partitioning (CP) is an easy combinatorial optimization problem that can be solved by either exact polynomial algorithms (e.g. dynamic programming algorithm DPA) [11], [12] or approximation algorithms (greedy type heuristics) of linear complexity. On the other hand, non-contiguous partitioning (NCP) is a hard scheduling problem for which several heuristics have been designed.

Several works have been devoted to the SMP problem. Bader and Heinecke [13] presented cache-oblivious algorithms based on space filling curves, with their high-performance shared memory implementations. In terms of multi-node parallelism, the literature is even sparser. Lugowski [14] presented parallel sparse matrix-matrix multiplication algorithm in a shared memory environment using a quadtree decomposition. Koanantakool et al [1] used different approaches for replicating data and partitioning work to minimize communication costs on over 10,000 cores. Moreover, CombBLAS adopts the SUMMA algorithm for AXPY kernel-product-parallel SpGEMM [15] and applies SUMMA which uses 2D partitioning of both the two input matrices. Besides, Ballard et. al. [8], recently provide lower bounds on the expected communication cost of parallel SpGEMM operation and propose two three-dimensional (3D) algorithms.

Our aim here is to study the parallel SMP where only one matrix is sparse. We focus on the problem of sparse matrix distribution leading to an optimal load balancing.

The remainder of the paper is organized as follows. In section II, we firstly recall some useful concepts. We then

detail different sparse partitioning approaches. We present two partitioning modes i.e. contiguous and non contiguous partitionings. Afterwards, we detail a particular complexity analysis of the designed algorithm. Section III is devoted to an experimental study achieved on two input data sets : (i) randomly generated input matrices (ii) a set of benchmark matrices from real applications. Finally, we conclude our work and propose some further perspectives.

II. GENERAL CONCEPTS

A. Sparse Matrices &Storage Formats

A matrix is called sparse if it has a large (resp. small) number of zero (resp. nonzero) elements [16]. Let nnz be the number of these latters. The density (%) of a sparse matrix of size, say N, is defined by $d = 100*nnz/N^2$. Processing sparse matrices requires using particular storage formats (SF) restricted to the nonzero elements such as DNS (DeNSe), CSR (Compressed Sparse Row) and COO (COOrdinate) [16].

B. Body kernel

The body kernel of a matrix product denoted C=AB, is based on various operations on vector elements [17]. There are three kind of body kernels i.e. DOT, AXPY, GAXPY [17].We recall that there are two variants of these body kernels depending on the access mode to matrix C, namely Row denoted R and Column denoted Col (see Figure 1):

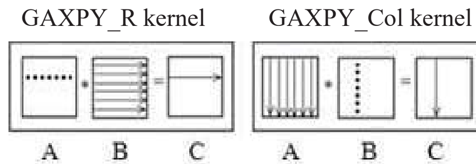

Fig 1. GAXPY kernels -Matrix access

III. THEORETICAL STUDY

Let us consider the parallelization of sparse dense matrix product (denoted SDMP) i.e. C = AB where A is an N×N sparse matrix, and both B and C are N×N dense matrices. Let us recall that the SDMP algorithm (denoted C=AB) requires 2N∗nnz flops. In this paper, we particularly focus on the GAXPY-Row body kernel where the matrices are acceded row-wise (Fig. 1).

Using N cores (or processors), row-wise matrix partitioning is trivial (see Fig. 1). As a matter of fact, each core multiplies a row A_i of A by corresponding rows of B to obtain row C_i of C. However, when p<N processors are available, we have to solve particular constrained scheduling and load balancing problems consisting in decomposing A into p row fragments (blocks) such that the weight of the fragment of maximal weight, equal to the scheduling makespan, is minimized, the weight being the total number of nonzero elements involved in the fragment. We'll use the term makespan henceforth.

Hence, using GAXPY-Row body kernel (see Fig.1 and (1)), partitioning SMP turns out to partitioning matrix A. Here, each nonzero element A_{lk} of the l-th row of A, denoted $A_{l.}$, is multiplied by the k-th row of B, denoted B_k. The l-th row of C, denoted $C_{l.}$, is thus computed as follows :

$$C_{l.} = \sum_{k=1}^{nzl} A_{lk} * B_{k.} \quad (1)$$

Where nzl is the number of nonzero elements in the l-th row of A. So, after applying load balancing techniques, each processor, among the p, computes some rows of C. Let us denote by NR_i the number of rows of matrix A assigned to the i-th processor P_i (i=1…p) and $Nnzp_i$ the total number of nonzero elements in these NR_i rows. The load $\mathcal{L}(i)$ of P_i i.e. number of operations it performs is equal to $2N*Nnzp_j$. Obviously, minimizing the maximal load is equivalent to minimizing the maximal weight. Henceforth, we'll consider these two terms as synonymous and the load will correspond to the weight. .

A. Data partitioning

In the following, we present various alternatives for partitioning a sparse matrix A into, say p, row fragments (blocks) where each fragment involves either contiguous [8], [11], [18], [19] or noncontiguous rows[7], [20]–[23].

1) Contiguous partitioning (CP)

CP consists in decomposing the matrix into row blocks (fragments) of same size i.e. involving the same number of rows (i.e either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$) [7], [8], [19]. Starting from such CP, Hamdi et al. proposed in [8] an improving approximation algorithm called GFSNR (Generalised Fragmentation with Same Number of Rows) and consisting in reducing the maximal load and increasing the minimal one by row transfer between successive fragments. This permits, through successive iterations, to refine the former fragmentation, hence leading to a better balanced one.

Salhi et al. presented in [19] an exact binary search algorithm of pseudo-linearithmic complexity. Their approach involves two phases. The first one consists in determining a bounding interval for the searched partitioning makespan (maximal load). They then proceed to construct successive partitionings through a binary search procedure. This latter may be either static or dynamic.

Anily and Federgruen [12] proposed an exact dynamic programming algorithm (DPA) of $O(N^2p)$ complexity. Olstad and Manne [11] improved the previous DPA and leading to an O(p(N-p)) complexity.

We propose in the following an alternative approximation algorithm called Nnz Balanced Partitioning (NBP).

Nnz Balanced partitioning (NBP)

Let w_j be the load (or weight) i.e. number of non zero elements of the j-th fragment (j=1…p), nz_i the number of nonzero elements in the i-th row of matrix A (i=1…N) and c the mean load i.e. $c = \lfloor nnz/p \rfloor$.The basic idea is as follows. We begin by assigning to the first fragment a number of rows such that its weight i.e. the total number of non zero elements of its rows is the closest to c without exceeding it. The row following the lastone of this fragment is added either to it or to its follower according to a dynamically updated ratio. The following fragments are processed in a similar manner. This

balancing procedure is applied to all but the last fragment to which all the remaining rows will be assigned. It is easy to notice that any fragment involves at least 1 row and at most N-p+1 rows.

We illustrate the algorithm by the following example.

n=10, p=3, the number of nonzero elements per row are :

nz	5	3	10	6	2	8	5	7	7	4
	1	2	3	4	5	6	7	8	9	10

nnz=57, c=57/3=19. The 1st fragment has rows 1, 2, 3 so $w_1=18<c$. If we add row 4, we get $w_1+nz_4=18+6=24>c$. Thus, we compare the two following ratios :

- $r1= |(w_1+nz_4-c)/nz_4|= |(18+6-19)/6|=|5/6|=0.833$
- $r2= |(w_1-c)/nz_4|= |18-19/6|=|-1/6|=0.166$

Since r2>r1, row 4 is assigned to fragment 2 (if we had r2≤r1, the row would be assigned to fragment 1). By iterating the procedure we finally obtain the following partitioning:

nz	5	3	10	6	2	8	5	7	7	4
	1	2	3	4	5	6	7	8	9	10
		$w_1=18$			$w_2=21$				$w_3=18$	

NBP algorithm writes as follows.

Algorithm 1 : NBP(nz)

```
wj=0 : j=1…n ; c=⌊nnz/p⌋; j=1;i=0 ;
WHILE (i≤N) DO
    i=i+1
    IF ((wj+nzi≤c) AND (i≤N-p+j)) THEN
        wj=wj+nzi
    ELSE IF (( i≤N-p+j) AND(j<p))THEN
            IF (|(wj+nzi-c)/nzi|≤|(wj-c)/nzi|)THEN
                wj=wj+ nzi
            ELSE
                wj+1=wj+1+ nzi
            ENDIF
            j=j+1
        ELSE IF((j=p) AND (i≤N))THEN
            wj=wj+ nzi
            ELSE IF (j<p)THEN
                DO k=i,N
                    j=j+1
                    wj=wj+ nzk
                ENDDO
                i=N
            ENDIF
        ENDIF
    ENDIF
    ENDIF
ENDWHILE
```

Clearly, algorithm NBP has an O(N) complexity.

2) Non ContiguousPartitioning (NCP)

NCP consists in decomposing A into balanced fragments involving non contiguous rows.

We find in the literature that the proposed heuristics for solving this NP-difficult scheduling problem are generally classified into two categories i.e. constructive heuristics and iterative improvement heuristics [24]. It turns out that the majority of the known heuristics belong to the first category where we may cite the list scheduling family including the well-known Longest Processing Time (LPT) algorithm [22], [24], [25].

As far as improvement heuristics are concerned, several methods have been proposed [7], [21]. An 0/1 interchange algorithm (with a complexity of O(Nlogp)) proposed in [21], proceeds by transferring or interchanging jobs (rows for us) between the most loaded processor (fragment to us) to the less loaded one. In [26], a descent algorithm using a critical reassign and a critical swap (interchange) neighborhood is proposed. It involves two phases. The first phase uses linear programming to construct a partial schedule while the second one applies an improvement procedure.

The different approaches we propose here are in fact variants of the S_GFSNZ approach [7] which corresponds to a two-phase heuristics. The first phase consists in sorting the rows of A in increasing number of non-zero elements (which may be done in an $O(N\log_2 N)$ time) then applying the classical (constructive) Longest Processing Time (LPT) heuristic [25]. The second phase, an improving one, is an iterative heuristic. It permits, through successive iterations, to refine the former fragmentation, hence leading to a better balanced one. This is done through successive row transfers then interchanges between the maximally and minimally loaded fragments.

Let Cmax (resp. Cmin) be the load of the most (resp. least) loaded processor denoted Pmax (resp. Pmin).

a) Transfer

Let σ = Cmax-Cmin, called load gap, and e the number of non zero elements of the row to be transferred from Pmax to Pmin. transferred. After this operation, the load gap becomes equal to σ' where :

$$\sigma' = (Cmax - e) - (Cmin + e) = \sigma - 2e$$

Two alternatives are possible here : (i) a feasible transfer where any row for which $1 \le e < \sigma$ may be chosen thus leading to a lower load gap ; (ii) an optimal transfer where the row chosen is such that e is the closest to σ/2 thus leading to an almost null load gap.

b) Interchange

Let us denote by e and e' (where e>e') the number of non zero elements of the two rows to be interchanged. After the interchange operation, the load gap is equal to:

$$\sigma' = (Cmax - e + e') - (Cmin + e' + e) = \sigma - 2(e-e')$$

Two alternatives are thus possible : (i) a feasible interchange where any couple of rows for which $1 \le e-e' < \sigma$ may be chosen thus leading to a lower load gap ; (ii) an optimal interchange where the chosen rows are such that e-e' is the closest to σ/2 thus leading to an almost null load gap.

As a matter of fact, several versions may be designed here where transfer or interchange are either exclusively used or mixed. We present in the following a series of 10 algorithms based on feasible/optimal transfer and/or feasible/optimal interchange where we proceed iteratively as long as the load gap criterion may be reduced or a fixed number of iterations is not exceeded. The 10 versions are the following : Feasible

Transfer (FR), Optimal Transfer (OT), Feasible Interchange (FI), Optimal Interchange (OI), Feasible Transfer-Interchange (FTI) (i.e. the S_GFSNZ proposed in [7]), Feasible Interchange-Transfer (OIT), Optimal Transfer-Interchange (OTI), Optimal Interchange-Transfer (OIT), MixFTI, MixOTI (see Algorithm 2 below). We have to mention that since the improving heuristics are iterative, we use a parameter Nit (number of iterations) as a termination criterion.

Algorithm 2 :MixOTI

```
WHILE ((σ ≥2)  AND (Nit ≤ σ) AND (change = true)) DO
|   Determine Pmax and Pmin; calculate σ          // O(N)
|   Search in Pmax the best element to be transferred (e ≈σ/2)
|   Compute the new makespan MT after transfer// O(N)
|   Search the best couple to be interchanged
|   (e-e'≈σ/2) // O(N²)
|   Compute the new makespan MS after interchange
|   IF (MT< MI) THEN
|   |    Transfer (e)
|   ELSE
|   |    Interchange (e,e')
|   ENDIF
ENDWHILE
```

c) Complexity

The following table recapitulates the complexities of the different versions for contiguous and non contiguous partitioning.

TABLE I. RECAPITULATION TABLE

Partitioning	Approach	Complexity
Contiguous	DPA[20], [21]	$O(pN^2)$
	DPA_Olstad [19]	$O(p(N-p))$
	GFSNR [15]	$O(N*Nit)$
	NBP	$O(N)$
Non contiguous	OT	$O(N*(\log N+Nit))$
	OI	$O(N^2*Nit)$
	OTI	$O(N^2*Nit)$
	OIT	$O(N^2*Nit)$
	MixOTI	$O(N^2*Nit)$
	FR	$O(N*(\log N+ Nit))$
	FI	$O(N^2*Nit)$
	FTI (S_GFSNZ [15])	$O(N*(\log N+ Nit))$
	FIT	$O(N*(\log N+ Nit))$
	MixFTI	$O(N^2*Nit)$

Since the number of iterations Nit depends on σ ($Nit \leq \sigma$) which depends on the input data, we have in most cases algorithms of *pseudo-polynomial complexity* [23].

IV. EXPERIMENTAL STUDY

A series of experimentations is achieved in order to evaluate the practical performances of our algorithms and validate our theoretical study. For this purpose, we used two input data sets i.e. (i) randomly generated input matrices (ii) a set of benchmark matrices from real applications. We henceforth denote by d the density of A.

We mention that we used an i7 Intel workstation (2.4 GHz clock, 8 GB RAM, 64Ko L1 cache, 256 KO L2 cache and 8 MO L3 cache) under Ubuntu OS. Our algorithms were coded in C. The run times correspond to the mean of several runs. We present in the following some excerpts of the different results (similar results were obtained with the other values) including a comparative study. We used the following notations :

- *M1: makespan of NBP*
- *M2 (resp.M3) : makespan of GFSNR (resp. DPA)*
- *RRR : Relative Reducing Ratio*
- $RRR_{1/2}=(|M1-M2|/Max(M1,M2))*100$
- $RRR_{1,2,3}=(|Min(M1,M2)-M3|/Min(M1,M2))*100$
- $RRR1 =((M1-M3) /M1)*100$
- $RRR2=((M2-M3)/ M2)*100$
- $RRR'=Cmax_v - Cmax_OI/Cmax_v)*100$
- *diff 1=M1- M3*
- *diff 2=M2 - M3*

where Cmax_v is the makespan of any other version.

1) Results for Contiguous Partitioning

NBP (resp. GFSNR) gives an optimal load balancing (i.e. the same makespan given by DPA) in 15% (resp. 2%) of the cases. For the remaining cases, the relative reducing ratio $RRR_{1/2}$ is within the interval [0.0003 1.67] (%). More precisely, NBP outperforms GFSNR in about 61% of cases (see Table II and Fig. 2).

TABLE II. MAKESPAN EVALUATION OF CONTIGUOUSPARTITIONING (N= 100000 AND D=0.05)

p	DPA	NBP	GFSNR	diff1	diff2	RRR1(%)	RRR2(%)
2	2500015	2500015	2500015	0	0	0	0
4	1250020	1250022	1250020	2	0	0.0002	0
8	625023	625023	625031	0	8	0	0.0013
16	312519	312521	312533	2	14	0.0006	0.0045
20	250021	250021	250031	0	10	0	0.0040
32	156279	156298	156287	19	8	0.0122	0.0051
40	125022	125022	125042	0	20	0	0.0160
60	83357	83357	83372	0	15	0	0.0180
80	62527	62530	62537	3	10	0.0048	0.0160
100	50025	50032	50047	7	22	0.0140	0.0440
120	41692	41695	41711	3	19	0.0072	0.0456
140	35740	35803	35757	63	17	0.1760	0.0475
160	31276	31283	31306	7	30	0.0224	0.0958
180	27803	27805	27823	2	20	0.0072	0.0719
200	25025	25196	25053	171	28	0.6787	0.1118

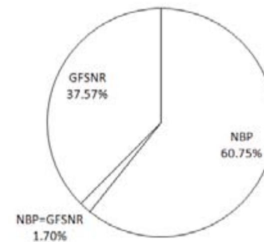

Fig. 2. Percentage of first ranked algorithms

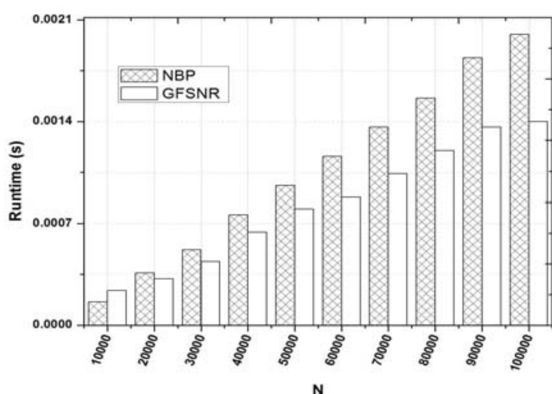

Fig. 3. Runtime of contiguousapproaches for randomly generated matrices d=0.05 and p=200

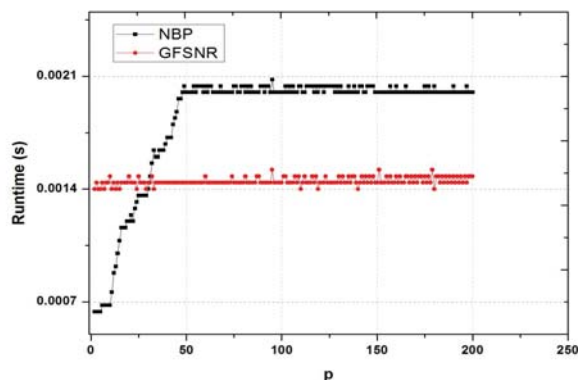

Fig. 4. Runtime of contiguous approaches for randomly generated matrix when N= 100000 and d=0.05

TABLE III. MAXIMUM AND MINIMUM LOAD OF NON CONTIGUOUS PARTITIONING (N=30000 AND D=0.5)

	p	2	4	8	16	20	40	60	80	100	120	140	160	180
LPT	**Cmax**	2250049	1125028	562514	281259	225008	112505	75004	56254	45004	37504	32224	28184	25042
	Cmin	2249951	1124923	562409	281151	224898	112393	74892	56140	44889	37388	32111	28069	24924
OI Optimal Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32164	28126	25011
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32140	28123	24983
	Nit	1	4	10	21	22	43	52	61	74	87	114	116	119
FI Feasible Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37501	32203	28175	25037
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37499	32115	28075	24935
	Nit	13	18	23	22	37	59	72	83	108	117	114	116	119
OT Optimal Transfer	**Cmax**	2250049	1125028	562514	281259	225008	112505	75004	56254	45004	37504	32224	28184	25042
	Cmin	2249951	1124923	562409	281151	224898	112393	74892	56140	44889	37388	32111	28069	24924
	Nit	1	1	1	1	1	1	1	1	1	1	1	1	1
FT Feasible Transfer	**Cmax**	2250049	1125028	562514	281259	225008	112505	75004	56254	45004	37504	32224	28184	25042
	Cmin	2249951	1124923	562409	281151	224898	112393	74892	56140	44889	37388	32111	28069	24924
	Nit	1	1	1	1	1	1	1	1	1	1	1	1	1
MixOTI Mix Optimal Transfer Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32164	28126	25011
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32140	28123	24983
	Nit	1	4	10	21	22	43	52	61	74	87	114	116	119
MixFTI Mix Feasible Transfer Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37501	32203	28175	25037
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37499	32115	28075	24935
	Nit	13	18	23	22	37	59	72	83	108	117	114	116	119
OTI Optimal Transfer Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32164	28183	25011
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32140	28069	24983
	Nit	1	4	10	21	22	43	52	61	74	87	114	116	119
FTI≡ S_GFSNZ Feasible Transfer Interchange	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37501	32203	28175	25037
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37499	32115	28075	24935
	Nit	13	18	24	22	37	59	72	83	108	117	114	116	119
OIT Optimal Interchange Transfer	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32164	28126	25011
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37500	32140	28123	24983
	Nit	1	4	10	21	22	43	52	61	74	87	114	116	119
FIT Feasible Interchange Transfer	**Cmax**	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37501	32203	28175	25037
	Cmin	2250000	1125000	562500	281250	225000	112500	75000	56250	45000	37499	32115	28075	24935
	Nit	13	18	23	22	37	59	72	83	108	117	114	116	119

TABLE IV. RANKING RATIO ACCORDING TO THE RUNTIME OF NON CONTIGUOUS PARTITIONING (%)

Rank	LPT	OI	FI	OT	FT	MixOTI	MixFTI	OTI	FTI(S_GFSNZ)	OIT	FIT
1	0.73	99.94	33.30	0.78	0.73	92.68	33.30	73.85	33.24	97.43	33.30
2	0.11	0.00	0.34	0.11	0.11	2.51	0.34	1.17	0.34	2.51	0.34
3	0.00	0.06	0.06	0.00	0.00	0.06	0.06	0.00	0.06	0.06	0.06

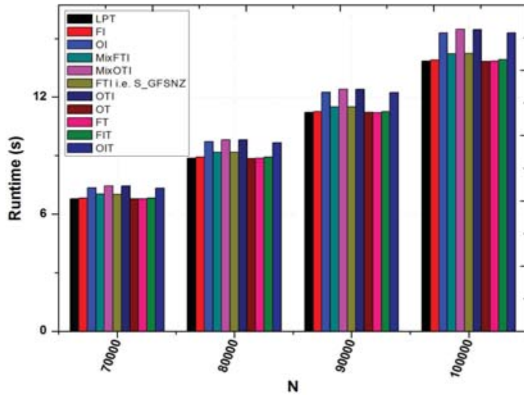

Fig. 5. Runtime of non contiguous partitioning using randomly generated matriceswhen d=1 and p=200

In the remaining 37% of cases, NBP is worse than GFSNR by about [5$*10^5$ 10] using $RRR_{1/2}$ ratios (%). These cases are essentially due to assigning the remaining rows to the last processor after load balancing of the p-1 first processors. We can provide an improvement step that allows spreading the overload on the other processors.

Using $RRR_{1,2,3}$ ratio, DPA is better than NBP and GFSNR about 0.16%.

However, NBP and GFSNR approaches are faster than DPA about 3 to 261 times. This ratio increases with p.

NBP is faster than GFSNR for p<30 (see Fig. 3 and Fig. 4).

2) Results of non contiguous partitioning

For small densities (d=0.05), LPT (phase 1) without amelioration phase gives an optimal makespan (σ=Cmax-Cmin$\leq$1). Using real matrix, LPT does not need an amelioration phase for almost of time since real matrix are hypersparse.

Makespan evaluation

OI (resp. OIT, MixOTI and OTI) partitioning gives often the best makespan i.e. in 99% (resp. 97%, 92%, 73%) of the cases (see Table IV). These versions provide improvement with a ratio (%) in the range [0.05 1.6] compared to the remaining versions (see Table V).

We hence remark that OI (resp. OIT, MixOTI and OTI) is followed by OT,FT, MixFTI, S_GFSNZ, then FI and OIT.

Notice that it is often impossible to apply transfer technique. However, interchange technique is usually more adequate. Indeed, finding a row to transfer from Pmax to Pmin seems impossible since all $nz_i \geq \sigma$. That's why, Nit$\leq$1 in most of time using FT and OT.

Run time evaluation :

MixOTI is the last ranked algorithm according to its runtime (see Fig. 5). However, FT and FI are generally the fasters since looking for a feasible solution costs less than looking for an optimal solution (see Table II).

Notice that for some cases, we have a unique solution i.e. feasible solution is equal to optimal solution.

TABLE V. RATIO RRR' EVALUATION OF NON CONTIGUOUS PARTITIONING (%) (N=30000 AND D=0.5)

p	LPT	OI	FI	OT	FT	MixFTI	OTI	FTI ≡ S_GFSNZ	OIT	FIT
2	0.0022	0	0	0.0022	0.0022	0	0	0	0	0
4	0.0025	0	0	0.0025	0.0025	0	0	0	0	0
8	0.0025	0	0	0.0025	0.0025	0	0	0	0	0
20	0.0036	0	0	0.0036	0.0036	0	0	0	0	0
60	0.0053	0	0	0.0053	0.0053	0	0	0	0	0
100	0.0089	0	0	0.0089	0.0089	0	0	0	0	0
120	0.0107	0	0.0027	0.0107	0.0107	0.0027	0	0.0027	0	0.0027
140	0.1862	0	0.1211	0.1862	0.1862	0.1211	0	0.1211	0	0.1211
160	0.2058	0	0.1739	0.2058	0.2058	0.1739	0.2022	0.1739	0	0.1739
180	0.1238	0	0.1038	0.1238	0.1238	0.1038	0	0.1038	0	0.1038
199	0.0221	0	0.0044	0.0221	0.0221	0.0044	0	0.0044	0	0.0044

FT , OT and FI is [0.7 2.7] (%) faster than FIT, [2.7 6.9] (%) better than MixFTI and FIT ≡ S_GFSNZ, [2.7 9.5] (%) better than OI and OIT, [4.42 10.65] (%) faster than MixOTI and OTI.

V. CONCLUSION

In this paper, we studied the sparse matrix partitioning for the SMP parallelization. Efficient partitioning approaches have been applied and led to interesting improvements. Indeed, NBP approach (which consists in partitioning the sparse matrix into blocks with contiguous rows) gives good performance in linear time. NBP is close to DPA which gives an optimal partitioning in a quadratic time.

Moreover, ten non contiguous approaches (i.e. consist in partitioning the sparse matrix into blocks with non contiguous rows) have been compared and divided into three groups : (i) approaches optimising makespan in quadratic time as OI, MixOTI, OIT (ii) approaches optimising time with non well balanced load as OT, FT (iii) approaches giving moderate balanced load in moderate time as FI, FIT, MixFTI, FTI i.e. S_GFSNZ, FIT.

Furthermore, the non contiguous approaches (with a complexity of at least O(N$*$log(N)+Nit)) give better makespan compared to contiguous approaches (with a complexity of O(N)).

To conclude, we can say that our work arises some interesting points constituting a further step in our work we intend to study in the near future. We may cite studying the parallelisation of Sparse Dense Matrix Product (SDMP), Dense Sparse Matrix Product (DSMP) and Sparse Matrix Vector Product (SMVP) using these partitioning approaches on different programming environments as many cores and GPU.

REFERENCES

[1] P. Koanantakool *et al.*, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 842–853.

[2] S. Acer, O. Selvitopi, and C. Aykanat, “Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems,” *Parallel Comput.*, vol. 59, pp. 71–96, Nov. 2016.

[3] D. Zheng, D. Mhembere, V. Lyzinski, J. Vogelstein, C. E. Priebe, and R. Burns, “Semi-external memory sparse matrix multiplication on billion-node graphs in a multicore architecture,” *CoRR*, vol. abs/1602.02864, Feb. 2016.

[4] K. Akbudak and C. Aykanat, “Exploiting locality in sparse matrix-matrix multiplication on many-core architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2258–2271, Aug. 2017.

[5] S. Ezouaoui, O. Hamdi-Larbi, and Z. Mahjoub, “Towards efficient algorithms for compressed sparse-sparse matrix product,” in *Proceedings of the 15th International Conference on High Performance Computing Simulation*, Genoa, Italy, 2017, pp. 651–658.

[6] K. Akbudak, O. Selvitopi, and C. Aykanat, “Partitioning models for scaling parallel sparse matrix-matrix multiplication,” *ACM Trans Parallel Comput*, vol. 4, no. 3, p. 13:1–13:34, Jan. 2018.

[7] O. Hamdi-Larbi, Z. Mahjoub, and N. Emad, “Load balancing in pre-processing of large-scale distributed sparse computation,” in *Proceedings of the 8th LCI Conference on High Performance Clustered Computing*, South Lake Tahoe, California, USA, 2007.

[8] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, “Brief announcement: hypergraph partitioning for parallel sparse matrix-matrix multiplication,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2015, pp. 86–88.

[9] S. Ezouaoui, Z. Mahjoub, L. Mendili, and S. Selmi, “Performance evaluation of algorithms for sparse-dense matrix product,” *Lect. Notes Eng. Comput. Sci.*, vol. 2202, no. 1, pp. 257–262, Mar. 2013.

[10] S. Ezouaoui, O. Hamdi-Larbi, and Z. Mahjoub, “Dense-sparse matrix multiplication : algorithms and performance evaluation,” in *Proceedings of the International Conference on Automation, Control, Engineering and Computer Science*, Sousse, Tunisia, 2014, pp. 87–94.

[11] B. Olstad and F. Manne, “Efficient partitioning of sequences,” *IEEE Trans. Comput.*, vol. 44, no. 11, pp. 1322–1326, Nov. 1995.

[12] S. Anily and A. Federgruen, “Structured partitioning problems,” *Oper. Res.*, vol. 39, no. 1, pp. 130–149, Feb. 1991.

[13] M. Bader and A. Heinecke, “Cache oblivious dense and sparse matrix multiplication based on peano curves,” *PARA*, vol. 8, 2008.

[14] A. Lugowski and J. R. Gilbert, “Efficient sparse matrix-matrix multiplication on multicore architectures,” in *CSC14: The 6th SIAM Workshop on Combinatorial Scientific Computing*, 2014, vol. 35.

[15] A. Buluç and J. R. Gilbert, “The combinatorial BLAS: design, implementation, and applications,” *Int J High Perform Comput Appl*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[16] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[17] D. Trystram and M. Cosnard, *Algorithmes et architectures parallèles*. Paris: Intereditions, 1993.

[18] M. M. A. Patwary *et al.*, “Parallel efficient sparse matrix-matrix multiplication on multicore platforms,” in *High Performance Computing*, 2015, pp. 48–57.

[19] H. Salhi, B. B. Mabrouk, and Z. Mahjoub, “An exact pseudo-linearithmic binary search algorithm for scheduling independent tasks under contiguity constraint,” in *Proceedings of the International Conference on High Performance Computing Simulation*, 2017, pp. 373–380.

[20] O. Tuncer, V. J. Leung, and A. K. Coskun, “PaCMap: Topology mapping of unstructured communication patterns onto non-contiguous allocations,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, New York, NY, USA, 2015, pp. 37–46.

[21] G. Finn and E. Horowitz, “A linear time approximation algorithm for multiprocessor scheduling,” *BIT Numer. Math.*, vol. 19, no. 3, pp. 312–320, Sep. 1979.

[22] L. Dipak and D. Kumar, “A Comprehensive Review and Evaluation of LPT, MULTIFIT, COMBINE and LISTFIT for Scheduling Identical Parallel Machines,” *Int J Inf Commun Techol*, vol. 11, no. 2, pp. 151–165, Jan. 2017.

[23] V.V. Vazirani, *Approximation Algorithms*. Springer-Verlag, 2001.

[24] S. P. Chen, Y. He, and G. Lin, “3-Partitioning Problems for Maximizing the Minimum Load,” *J. Comb. Optim.*, vol. 6, no. 1, pp. 67–80, Mar. 2002.

[25] R. L. Graham, “Bounds for certain multiprocessing anomalies,” *Bell Syst. Tech. J.*, vol. 45, no. 9, pp. 1563–1581, Nov. 1966.