A Parallel Sloan Algorithm for the Profile Minimization Problem of Sparse Matrices

Thiago Nascimento Rodrigues
Maria Claudia Silva Boeres
Lucia Catabriga
Federal University of Espírito Santo
Av. Fernando Ferrari, 541, 29075-910, Vitória, ES, Brasil
{tnrodrigues,boeres,luciac}@inf.ufes.br

ABSTRACT

The Sloan algorithm is a well-know heuristic for reordering sparse matrices. It is typically used to speed up the computation of sparse linear systems of equations. This paper presents an OpenMP parallel Sloan version. The performance is compared with a serial implementation made available by Boost library. The concept of logical data structure is explored in order to leverage the parallelism on contiguous memory positions. The algorithm reached profile reduction rates up to 99.67%, and outstanding CPU time improvements were also achieved - up to 99.87%. Furthermore, speedup ratios up to 3.69X were achieved for a set of large sparse matrices.

KEYWORDS. Parallel Sloan. Profile Minimization. Sparse Matrices.

Paper topics (Parallel Algorithms on CPU&GPU for Operational Research)

1. Introduction

Computations involving sparse matrices have been of widespread use since the 1950s, and its applications include electrical networks and power distribution, structural engineering, reactor diffusion, and solutions to differential equations [Pooch and Nieder, 1973]. According to Saad [Saad, 2003], partial differential equations represent the biggest source of sparse matrix problems. The linear systems that arise from the discretization of these equations are of the type $Ax = b$, in which A is a large and sparse matrix, that is, it has very few non-zero (NNZ) entries.

The profile minimization problem (PMP) was originally proposed as an approach to reducing the space requirements for storing sparse matrices [Tewarson, 1973]. Additionally, the PMP enhances the performance of operations on systems of nonlinear equations. Most methods designed to reduce the matrix profile are based on the corresponding graph formulation: find a labeling of the vertices of a graph such that most connections are between vertices having close labels. However, Lin and Yuan [Lin and Yuan, 1994] proved that the PMP of an arbitrary graph is equivalent to the interval graph completion problem, which was shown to be NP-complete by [Garey and Johnson, 1990]. Hence, several heuristics and metaheuristics have been proposed for the problem. As example, recently a Scatter Search metaheuristic was presented by Sánchez-Oro et al. [Sánchez-Oro et al., 2015] as the state-of-the-art solution method for the PMP. Moreover, some of the most important and well-known heuristics for the problem are Reverse Cuthill-McKee [Cuthill and McKee, 1969] and Sloan [Sloan, 1986]. These two last ones use graph search strategies and provide high quality solutions.

Some parallelizations of solution methods for PMP have been developed in order to reach greater performance from multi-core processors. Lin [Lin, 2005] introduced a genetic parallel algorithm tailored to this problem, and [Karypis and Kumar, 1998] presented a parallel formulation of the multilevel graph partitioning and sparse matrix ordering problem. More lately, Karantasis et al. [Karantasis et al., 2014] developed a parallel implementation of the Sloan algorithm based on a thread-level speculation model. In this work, a non-speculative parallelization of this same algorithm is proposed. A concept of logical data structure is used in order to leverage the paralellism on contiguous memory positions. The performance of parallel Sloan is compared with a respective serial implementation made available by Boost library [Siek et al., 2002]. The parallelism is supported by the OpenMP framework[1] and a set of large sparse matrices is used to test the algorithms.

The rest of this paper is organized as follow. In Section 2, a review background information is provided and the original serial Sloan algorithm is also presented. Section 3 is dedicated to detail the non-speculative parallel Sloan algorithm proposed by this work. Section 4 presents the results from Boost library and parallel Sloan. Conclusions and directions for future work are included in Section 5.

2. Background

Let $A = [a_{ij}]$ be a structurally symmetric matrix[2], whose diagonal elements are all non-zero. For the i^{th} row of A, $i = 1, 2, \ldots, n$, let

$$f_i(A) = \min\{j \mid a_{ij} \neq 0\}, \text{ and } \beta_i(A) = i - f_i(A).$$

The number $f_i(A)$ is simply the column subscript of the first non-zero component in row i of A. The number $\beta_i(A)$ is called **row width** of the i^{th} row, and it is the difference between i and the column index of the first non-zero element on the i^{th} row. Since the diagonal entries a_{ii} are positive or non-null, $f_i(A) \leq i$ and $\beta_i(A) \geq 0$. The **bandwidth** of A is defined as

$$\begin{aligned}\beta(A) &= \max\{\beta_i(A) \mid 1 \leq i \leq n\}\\ &= \max\{|i - j| \mid a_{ij} \neq 0\}.\end{aligned}$$

[1]OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs Dagum and Menon [1998].

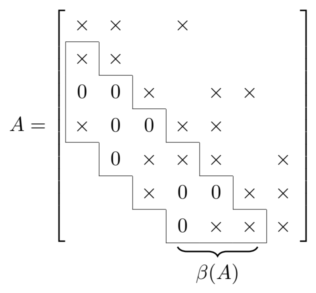
[2]A matrix $A = [a_{ij}]$ is said be structurally symmetric if $a_{ij} \neq 0$ then $a_{ji} \neq 0$, but not necessarily $a_{ij} = a_{ji}$.

The number $\beta_i(A)$ is called the i_{th} bandwidth of A. So, the **band** of A is defined as *Band(A)* $=$ $\{\{i,j\} \mid 0 < i - j \leq \beta(A)\}$, which is the region within $\beta(A)$ locations of the main diagonal. This region is delimited, in each row, by the columns i and j. The union of all sub-regions per row is denoted by {i, j}.

A slightly more sophisticated scheme for exploiting sparsity is the so-called **envelope** or **profile**, which simply takes advantage of the variation in $\beta_i(A)$ with i. The envelope of A, denoted by *Env(A)*, is defined by *Env(A)* $= \{\{i,j\} \mid 0 < i - j \leq \beta_i(A)\}$. In terms of the column subscripts $f_i(A)$, the relation is *Env(A)* $= \{\{i,j\} \mid f_i(A) \leq j < i\}$. The envelope of a structurally symmetric matrix is easily visualized: picture the lower triangle of the matrix, and remove the diagonal and the leading zero elements in each row. The remaining elements (whether non-zero or zero) are in the envelope of the matrix [Kumfert and Pothen, 1997]. The number of these elements defines the quantity —*Env(A)*— called **profile** or **envelope size** of A, and is given by

$$|\mathit{Env(A)}| = \sum_{i=1}^{n} \beta_i(A).$$

The matrix example in Figure 1 has a bandwidth of 3 and a profile of 11.



i	$f_i(A)$	$\beta_i(A)$
1	1	0
2	1	1
3	3	0
4	1	3
5	3	2
6	3	3
7	5	2
$\sum$		11

Figure 1: Example of bandwidth and profile.

2.1. Sloan Algorithm

The idea of Sloan's algorithm [Sloan, 1986] is to number vertices from one point of an approximate diameter in a unweighted graph, choosing the next vertex to number from among the neighbors of currently numbered vertices and their neighbors. A vertex of maximum priority is chosen from this eligible subset of vertices. Based on this, the Sloan algorithm may be described as a composition of two distinctive phases: (1) selection of a start vertex s and an end vertex e, and (2) vertex reordering. Step 1 looks for a pseudodiameter of the graph and choose s and e to be the endpoints of the pseudodiameter. In Step 2, the pseudodiameter is used to guide the reordering.

Each vertex of the graph is given a priority and the start vertex s is ordered first. Then, at each stage, the next vertex is chosen among eligible vertices with the highest priority. Thus, a balance is maintained between the aim of keeping the profile small and bringing in vertices that have been left behind (far away from e). The list of eligible vertices comprise those that are in the front (neighbors of one or more renumbered vertices) or neighbors of one or more vertices in the front [Hu and Scott, 2001].

Algorithm 1 gives a sketch of the serial Sloan. At each step, a node in the graph can be in one of these four states: (i) *numbered*; (ii) *active*; (iii) *preactive*, a non-numbered and non-active node that is a neighbor of an active node; and (iv) *inactive*, all other nodes. Initially the source node is preactive and all other nodes are inactive (lines 1-5). The algorithm iterates through all the nodes in the graph and at each step it chooses among the active or preactive nodes the one that maximizes the priority (lines 8-23). New priorities are assigned to the neighbors and their neighbors are selected (lines 24-31).

Algorithm 1 Serial Sloan Algorithm

```
Input: Graph G, Weight $W_1$, $W_2$, Node $s$, $e$
 1: foreach (Node n : G.nodes) {
 2:     n.status = inactive;
 3:     n.priority = $W_1$ * n.distance - $W_2$ * n.degree;
 4: }
 5: s.status = preactive
 6: Worklist wl = { s };
 7: nextId = 0;  P[nextId++] = $s$;
 8: foreach (Node n : wl)  ordered by (-n.priority) {
 9:     foreach (Node v : G.neighbors(n))  {
10:         if ( n.status == preactive $\wedge$ (v.status == inactive $\vee$ v.status == preactive)) {
11:             update(v.priority);
12:             v.status = active;
13:             updateFarNeighbors(v, wl);
14:         } else if (n.status == preactive $\wedge$ v.status == active)  {
15:             update(v.priority);
16:         } else if (n.status == active $\wedge$ v.status == preactive)  {
17:             update(v.priority);
18:             v.status = active;
19:             updateFarNeighbors(v, wl);
20:     }   }
21:     P[nextId++] = n;
22:     n.status = numbered;
23: }
24: function UPDATEFARNEIGHBORS(Node v, Worklist wl) {
25:     foreach (Node u : G.neighbors(v))  {
26:         if (u.status == inactive)  {
27:             u.status = preactive;
28:             wl.push(u);
29:         }
30:         update(v.priority);
31: }   }
```

3. A Parallel Sloan Algorithm

According to Karantasis et al. [Karantasis et al., 2014], the most natural parallelization of Sloan algorithm is to process multiple iterations of the outermost loop (lines 8-23) of the Algorithm 1 in parallel. However, this natural strategy presents two main challenges to the parallelization. First, the outer loop relies on a priority queue to determine the next active node to examine. Second, for each active node, a significant portion of the graph must be manipulated, and it is likely that two successive active nodes will have overlapping neighborhoods.

In order to work around these challenges, this work uses an alternative approach to guide the nodes traversing based on priority orders. The serial Sloan (Algorithm 1) as well as the speculative parallel Sloan described by Karantasis et al. [Karantasis et al., 2014], sort all active and preactive nodes by the priority. The best ranked nodes are processed and the respective priority and status of them are updated. The parallel Sloan implemented in this work, on the other hand, makes no use of a sorting operation through each iteration. Actually, it uses a concept of logical bags to drive the nodes processing. The bag data structure is an unordered collection of elements in which the order of insertion is completely irrelevant. Furthermore, elements can be inserted and removed entirely at random [Budd, 2016]. The idea behind logical bags employed in the implemented Sloan algorithm is to identify collections of nodes that may be seen as bag of priorities. In other words, instead of sorting nodes to process them in the correct priority order, our parallel Sloan algorithm keep a reference for each collection of nodes (logical bags) which are grouped by the priority. Thus, after determining the logical bag with highest priority, the respective elements are recovered and processed. If a node priority is updated, the corresponding node reference is updated accordingly.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nodes:	e	b	d	a	m	j	c	t	u	i		a	b	c	
priorities:	14	2	4	13	11	14	15	13	12	14		8	12	10	

	1	2	3	4	5	6	7	8	9	10
logical bags:	2	4	8	10	11	12	13	14	15	
size:	1	1	1	1	1	2	2	3	1	

Figure 2: Logical bags used by Parallel Sloan.

Figure 2 presents an example of the logical bags used by the parallel Sloan algorithm. Nine priority bags are identified in the array of nodes. Three of them are highlighted by colors: one of size 3 is shaded with gray, and other two bags of size two are shaded with pink and green. An auxiliary array of logical bags stores the size of each bag. It is updated after each node processing. In this work, the logical bags strategy was implemented as detailed by Algorithm 2. Each vertex of the graph is given a initial priority $P(i)$ (line 6) such that

$$P(i) = W_1\ distance(i, start\_node) - W_2\ degree(i)$$

where W_1, and W_2 are positive weights. These weights used to ponder the nodes priority were chosen following the recommendation reported by Sloan [Sloan, 1986] and highlighted by Reid and Scott [Reid and Scott, 2012]. Thus, the Bag-based Sloan implementation has used the pair (2, 1) as weights for the global priority function.

As the nodes priorities are stored in a static array, and the priorities generated by Sloan algorithm may be negative values, thus, firstly, the algorithm carries out a priority value shifting. For this change, the minimum priority is determined (lines 4-11) and it is used as offset for all other priority values (line 13). It is important to notice that the array of priorities is a two-dimensional array. The first row stores the current priority value and the second row stores the most updated value. Initially, both positions have the same value (line 14).

Algorithm 3 presents the Parallel Bag-based Sloan algorithm implemented in this work. First of all, a level structure is built through a Breadth-First Search procedure (line 1). As it is based on the concept of logical bags, its second step involves the generation of the initial set of priority bags. Algorithm 2 is invoked (line 2) for this task. The size of the logical bags array is overestimated using as base the priority of the start node and an empirical factor (line 3). As the number of bags must be determined before the use of them by the Bag-based Sloan (Algorithm 3), the $PRIORITY\_FACTOR$ constant constitutes a way to oversize the maximum number of bags. Several tests were made in order to identify a relation between the initial priority of the pseudo-peripheral start node ($start$ node) and the number of bags. No closed expression has been found associating these two variables. Nevertheless, taking the start node as input for a number of bags estimation, the set of executed tests have shown no wide variation between the expected value and the obtained value. This narrow variation made possible the definition of a factor to compose an overestimated computation for the size of the array of bags. In this way, the value of 10 has been assigned to the *PRIORITY_FACTOR* used in the Bag-based Sloan algorithm.

Algorithm 2 Parallel Priority Bags Generator Algorithm

```
Input: Graph G, Weight $W_1$, $W_2$
 1: int minPriority = INFINITY;
 2: int priority[G.numNodes][2];
 3: parallel {
 4:     int minThreadPriority = INFINITY;
 5:     foreach (Node n : G.nodes) {
 6:         priority[n][CURRENT] = $W_1$ * n.distance - $W_2$ * n.degree;
 7:         if (priority[n][CURRENT] < minThreadPriority)
 8:             minThreadPriority = priority[n][CURRENT];
 9:     }
10:     atomic if (minThreadPriority < minPriority)
11:         minPriority = minThreadPriority;
12:     foreach (Node n : G.nodes) {
13:         priority[n][CURRENT] -= min_priority;
14:         priority[n][NEW] = priority[n][CURRENT];
15: } }
16: return priority;
```

Each iteration executed by the outermost loop of the Algorithm 3 is composed by three stages. **(1) Choice of the maximum priority bag.** At lines 8 to 14, the set of priority bags references are analysed in order to identify the maximum non-empty bag. As each position of the array of logical bags corresponds to an specific bag reference, the array is iterated in a reverse order. Thus, as soon as a bag is found, the traversing is interrupted (line 13) and a maximum priority bag is identified (line 12). This step is executed by just one thread (serial algorithm region).

(2) Priority bag processing. This stage is executed by the loop at lines 15 to 39. It corresponds to the step of the serial Sloan in which vertices are numbered and its respective neighbors and their neighbors have their priorities and status updated. Naturally, all updates are guided by the sequence of four status proposed by the Sloan algorithm. It is important to notice that only nodes belonging to the maximum bag priority are processed in this stage. However, there is not an arrangement of nodes in order to group them into the appropriated max bag. This grouping of nodes is carried out by matching the priority of each one with the priority of the maximum bag chosen at the first stage (line 16).

(3) Bags size update. The last loop (lines 41-46) executes the update of each bag priority size. When a node priority is updated, this fact may be seen like a node going out from the previous priority bag to the new priority bag. As the algorithm uses just the concept of bag, i.e., it does not use any bag data structure, so these operations related to nodes coming in and coming out of priority bags are emulated by decreasing and increasing the size of the bags respectively (lines 43 and 44). Moreover, a priority node updating is identified by the comparison between the last modified priority (NEW position of priority array) and the current one (CURRENT position). If both are different, there was a priority updating (line 45).

Following a parallelization feature suggested by Karantasis et al. [Karantasis et al., 2014], the Bag-based Sloan algorithm proposed in this work also implements atomic operations at the node update level rather than at the whole iteration level. All status and priority updates are serialized by an *atomic* operator. Nevertheless, the whole iteration is executed in a parallel way. The same takes place during the bags size updating in the last stage. Naturally, when multiple threads update multiple nodes at the same time, the order of each update varies between an algorithm execution and another one. It leads to different permutation arrays, and, hence to different nodes reordering.

Algorithm 3 Parallel Bag-based Sloan Algorithm

```
Input: Graph G, Weight $W_1$, $W_2$, Node start, end
1: G = Breadth_First_Search(G, start);
2: int[][] priority = Priority_Bag_Generator(G, $W_1$, $W_2$);
3: int num_bags = PRIORITY_FACTOR * priority[start][CURRENT];
4: int[num_bags] bags;   start.status = PREACTIVE;   nextId = 0;
5: bags[priority[start][CURRENT]]++;
6: parallel {
7:     while (nextId < G.size) {
8:         serial { // Choosing maximum priority bag
9:             int max_priority = 0;
10:            foreach (int prior_bag : [num_bags, 0]) {
11:                if (bags[prior_bag] > 0) {
12:                    max_priority = prior_bag;
13:                    break;
14:        } } }
15:        foreach (Node n : G.nodes) { // Processing logical bag
16:            if (n.status $\neq$ NUMBERED $\wedge$ priority[n][CURRENT] == max_priority) {
17:                foreach (Node v : n.neighbors) {
18:                    update_far = FALSE;
19:                    if (n.status == PREACTIVE $\wedge$ (v.status == INACTIVE $\vee$ PREACTIVE) {
20:                        atomic priority[v][NEW] += $W_2$;
21:                        atomic v.status = ACTIVE;
22:                        update_far = TRUE;
23:                    } else if (n.status == PREACTIVE $\wedge$ v.status == ACTIVE) {
24:                        atomic priority[v][NEW] += $W_2$;
25:                    } else if (n.status == ACTIVE $\wedge$ v.status == PREACTIVE) {
26:                        atomic priority[v][NEW] += $W_2$;
27:                        atomic v.status = ACTIVE;
28:                        update_far = TRUE;
29:                    }
30:                    if (update_far) {
31:                        foreach (Node u $\in$ v.neighbors $\wedge$ u $\neq$ n) {
32:                            if (u.status == INACTIVE)
33:                                atomic u.status = PREACTIVE;
34:                            atomic priority[u][NEW] += $W_2$;
35:                } } }
36:                atomic {
37:                    bags[priority[n][CURRENT]]--;
38:                    P[nextId++] = n;   n.status = NUMBERED;
39:        } } }
40:        // Updating size of bags
41:        foreach (Node n $\in$ G.nodes $\wedge$ n.status $\neq$ NUMBERED) {
42:            if (priority[n][CURRENT] $\neq$ priority[n][NEW]) {
43:                atomic bags[priority[n][CURRENT]]--;
44:                atomic bags[priority[n][NEW]]++;
45:                priority[n][CURRENT] = priority[n][NEW];
46: } } } }
```

4. Experimental Results

As the source code of the original parallel algorithm is not available for public use, the performance evaluation of the parallel Bag-based Sloan was against a respective serial implementation made available by Boost Library [Siek et al., 2002]. A set of twenty structural symmetric and square matrices was selected from the University of Florida Sparse Matrix Collection [Davis and Hu, 2011]. These matrices cover multiple type of problems in order to increase the dataset variety and the percentage of sparsity of each one is higher than 99.90%.The set of tested matrices is shown in Table 1. The columns present matrices and some characteristics of them: size (number of rows/columns) and number of non-zeros.

Table 1: Sparse Tested Matrices.

#	**Matrix**	**Size**	**Non-zeros**	**#**	**Matrix**	**Size**	**Non-zeros**
01	m_t1	97,578	9,753,570	11	inline_1	503,712	36,816,170
02	filter3D	106,437	2,707,179	12	gsm_106857	589,446	21,758,924
03	SiO2	155,331	11,283,503	13	Fault_639	638,802	27,245,944
04	d_pretok	182,730	1,641,672	14	tmt_sym	726,713	5,080,961
05	CO	221,119	7,666,057	15	boneS10	914,898	40,878,708
06	offshore	259,789	4,242,673	16	audikw_1	943,695	77,651,847
07	Ga41As41H72	268,096	18,488,476	17	nlpkkt80	1,062,400	28,192,672
08	F1	343,791	26,837,113	18	dielFilterV2real	1,157,456	48,538,952
09	mario002	389,874	2,097,566	19	Serena	1,391,349	64,131,971
10	msdoor	415,863	19,173,163	20	G3_circuit	1,585,478	7,660,826

Table 2: Parallel Sloan Comparison - Profile and CPU time (sec.)

Matrix		Final Profile		Reordering Time		Threads	
#	Profile	Boost	Bag	Boost	Bag	#	Speedup
01	250,103,781	116,018,656	117,597,457	2.661	**0.102**	04	3.69
02	260,719,523	94,647,226	102,301,580	2.262	**0.025**	12	3.36
03	2,482,485,963	1,747,699,515	1,752,130,242	24.026	**0.047**	12	2.38
04	5,143,933,251	183,016,114	230,212,781	3.343	**0.013**	04	3.21
05	4,463,242,951	2,901,790,897	2,942,437,379	40.754	**0.058**	04	3.03
06	3,588,201,815	1,306,762,040	1,535,446,907	19.826	**0.080**	12	2.80
07	6,188,064,837	5,061,789,451	5,120,317,933	71.836	**0.094**	04	2.59
08	41,341,166,041	802,493,269	887,690,105	16.461	**0.109**	12	2.47
09	48,023,597,183	114,504,010	158,902,510	2.167	**0.084**	08	2.52
10	23,550,593,343	681,799,054	682,817,688	11.733	**0.306**	04	2.95
11	57,452,838,418	1,030,990,936	1,024,217,929	19.644	**0.217**	12	2.61
12	161,502,078,777	1,741,744,855	2,029,470,672	32.637	**0.365**	04	1.83
13	8,421,729,249	5,999,798,831	6,603,660,438	81.144	**0.908**	08	2.81
14	593,531,565	595,428,857	596,248,727	6.717	**0.598**	08	2.72
15	6,345,023,025	4,087,417,386	4,040,804,407	62.442	**3.205**	08	2.53
16	390,218,762,952	7,046,316,450	7,840,128,498	149.580	**0.242**	04	2.43
17	275,456,486,321	21,884,523,357	21,889,103,451	342.581	**0.983**	12	2.42
18	550,359,970,640	4,997,283,308	6,765,505,540	113.890	**0.287**	12	2.33
19	77,321,314,974	31,769,020,684	32,332,635,404	791.253	**3.786**	08	2.37
20	119,101,864,114	2,946,942,317	2,976,611,592	32.731	**1.343**	08	2.47

The program was coded in the C language and the parallelism was supported by OpenMP framework. The complete source code is available on a GitHub repository [Rodrigues, 2017]. The experiments were performed on a PC running Ubuntu Linux, version 14.04.5 LTS, with Kernel version 3.19.0-31. It consists of a Intel i7-3610QM processor of 4 cores operating at 2.3 GHz. Each core has a unified 256KB L2 cache and the processor has a shared 6MB L3 cache. The PC contains 8GB of main memory, the code was compiled with GNU Compiler Collection (GCC) version 5.4.0, and $-O3$ optimization flag was turned on. Moreover, the Compressed Sparse Row format (CSR) [Farzaneh et al., 2009] was the mechanism used to store each tested matrix. The operations applied on them were also performed using this format.

Table 3: Percentages of Final Profile Variation and CPU time Reduction - Boost x Bag Sloan

Matrix	Profile Var.	Time Red.	Matrix	Profile Var.	Time Red.
m_t1	-1.177	96.17	inline_1	+0.012	98.90
filter3D	-4.609	98.89	gsm_106857	-0.180	98.88
SiO2	-0.603	99.80	Fault_639	-24.933	98.88
d_pretok	-0.951	99.61	tmt_sym	†	91.10
CO	-2.603	99.86	boneS10	+2.065	94.87
offshore	-10.024	99.60	audikw_1	-0.207	99.84
Ga41As41H72	-5.197	99.87	nlpkkt80	-0.002	99.71
F1	-0.210	99.34	dielFilterV2real	-0.324	99.75
mario002	-0.093	96.12	Serena	-1.227	99.52
msdoor	-0.004	97.39	G3_circuit	-0.026	95.90

† denotes that there was no profile reduction.

Both algorithms Sloan from Boost library and the parallel Bag-based Sloan were performed five times for each pair (m_i, t_j), where m_i is a sparse matrix, and t_j is the number of threads between 1 and 12 (in steps of 2) used by each algorithm. For each (m_i, t_j) tested pair, the average was calculated from the reported values. In order to confront the algorithms, for each matrix m_i, it was selected the number of threads t_j that reached the best value considering the CPU time. Moreover, the speedup S computed for the parallel Sloan algorithm was calculated according to expression $S(n) = \frac{T_1}{T_n}$, where T_1 is the run-time of the parallel Bag-based Sloan executed with one thread, and T_n is the run-time of the same algorithm executed with n threads.

Table 2 presents a comparison of the algorithms performance according to the profile after reordering and CPU time. The two columns grouped under Threads indicate (i) the number of threads (column #) that produced the best CPU time, and (ii) the speedup ratio (column Speedup) reached with the related set of threads. For all tested matrices, the implemented parallel Sloan shown an outstanding reordering performance. In fact, the best CPU time was attained by Bag-based Sloan for the whole set of sparse matrices. Table 3 describes the percentage of the time reduction between the two evaluated algorithms (column Time Red.). As displayed by it, the parallel Bag-based Sloan achieved CPU time reductions ranging from 91.10% (tmt_sym) to 99.87% ($Ga41As41H72$) when compared with the Boost library implementation.

Even though the final profile produced by Boost library has been better than the implemented Sloan, the percentage of variation between the two algorithms was very slight. Indeed, Table 3 presents the variation ratio between the profile percentage reduction (column Profile Var.) reached by the algorithms. As can be seen in Table 3, only for two matrices - $Fault\_639$ and $offshore$ - the variation was significant - 24,933% and -10.024%, respectively. For all others tested matrices, the ratio was inferior to 5.2%. However, these differences are negligible when compared with the initial profile. Furthermore, for the matrices $inline\_1$ and $boneS10$, the percentage of profile variation was positive for the Bag-based Sloan algorithm, i.e., it has reached a better profile minimization than Boost library. An exception was the matrix tmt_sym - both algorithms were not able to produce any profile reduction.

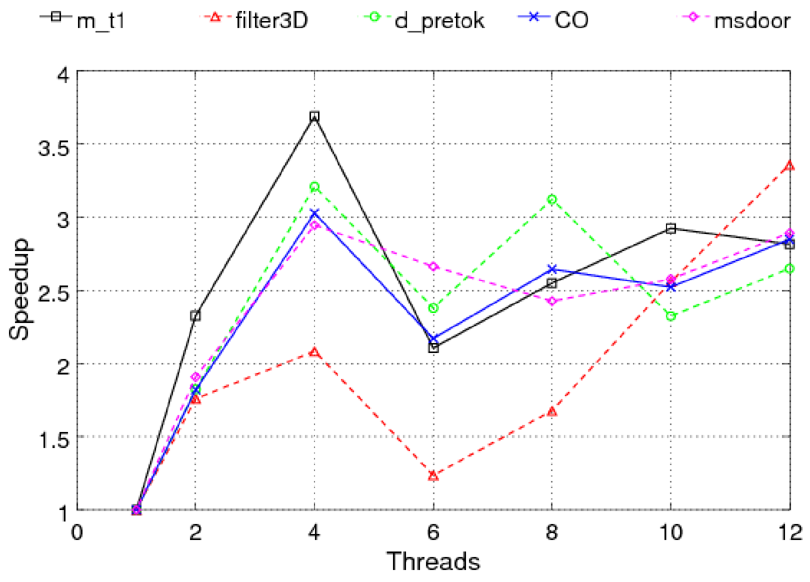


Figure 3: Bag-based Sloan speedup.

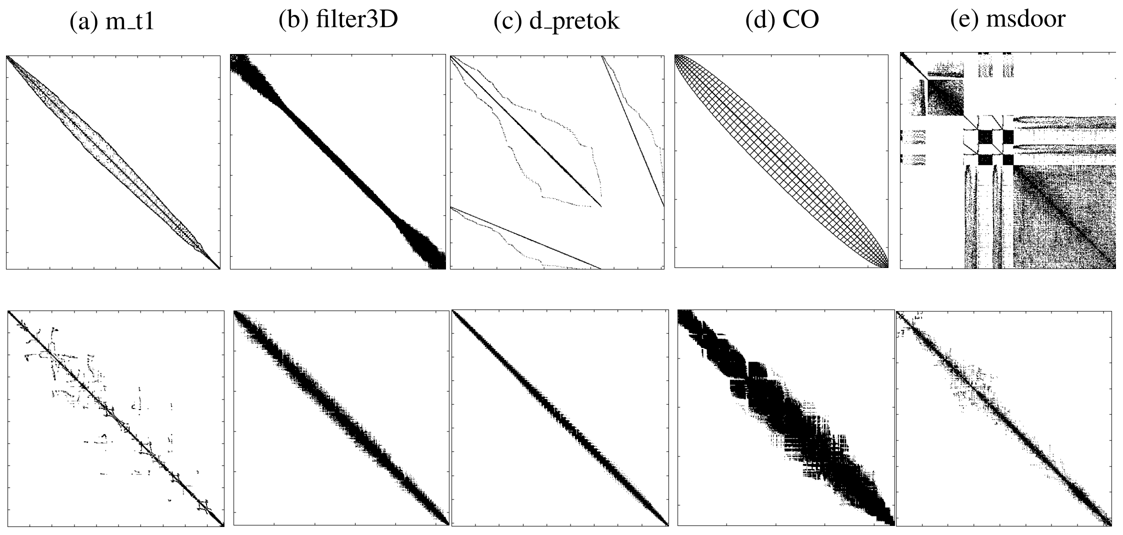


Figure 4: Profile pattern after reordering

The five highest speedups reached by Bag-based Sloan algorithm are displayed in Figure 3. As presented by the speedup graphic, there was a performance decrease when the five matrices were processed with six threads. Since the algorithm does not implements any specific load balance policy, every level of oversubscription may lead to a non-deterministic effect on the performance. Actually, running the algorithm with a number of threads higher than the number of available cores may (i) cause the Operating System (OS) continually move threads between cores in an effort to balance the load, or (ii) persuade the OS to give the OpenMP code a larger share of the CPU resources. In the first case, the performance can be compromised as observed when running the Bag-based Sloan algorithm with 6 threads. On the other hand, if the second behavior happens, a performance improvement can be seen as detected through the experiments with 8 and 12 threads.

Despite this unpredictable behavior when the algorithm is executed with oversubscribed threads, the tests with all aforementioned five matrices shown a relevant speedup. In fact, the implemented algorithm achieved a performance improvement ratio up to 3.69X (m_t1), 3.36X ($filter3D$), 3.21X (d_pretok), 3.03X (CO), and 2.95X ($msdoor$). In contrast with the original algorithm [Karantasis et al., 2014], the scalability of the proposed algorithm did not show a close relationship with the number of non-zeros per row of a tested sparse matrix. Actually, instead of traversing the graph guided only by a level structure, the Bag-based Sloan algorithm takes also into account the priority clusters of nodes (bags of nodes). Because of this, a graph with a large number of edges per node do not necessarily cause a speedup boosting. In fact, considering this set of five matrices, four of them (m_t1, $filter3D$, CO, and $msdoor$) present a high average of NNZ/row: 100, 25, 35, and 46, respectively. On the other hand, this is not the case of the matrix d_pretok, which has around only 9 NNZ/row. Additionally, the reordering quality attained by these same matrices can be attested by the profile pattern generated by them after applying the permutation. Figure 4 highlights graphically the Bag-bases Sloan efficiency for the profile minimization problem. The first row exhibits the original profile of each matrix. The respective column in the row below shows the sparsity pattern of the correspondent matrix after the reordering.

5. Conclusions

This paper presented a parallel version of Sloan heuristic whose nodes processing is controlled by a logical data structure. The algorithm performance and the reordering quality produced by it were compared with the results achieved by its respective serial version made available by Boost library. The implemented algorithm has improved the Boost CPU time up to 99.87%, and the highest speedup reached by it was of 3.69X running with four threads. The sparsity pattern obtained by applying the permutation generated by the algorithm also attests its efficiency. In fact, the reductions attained by it various from 91.10% to 99.87% of the original profile. Based on these performance outcomes, the Bag-based Sloan algorithm may be recommended as an efficient parallel heuristic for the profile minimization problem.

Some features have shown themselves to be heavy factors for the performance of Sloan algorithm. It is the case of the priority function and the underlying data structure used to support the nodes processing. Improved serial versions of the Sloan were already published with more elaborated priority functions [Kumfert and Pothen, 1997]. Furthermore, some works in the literature have addressed the reordering problem through the use of other data structures. As example, [Leiserson and Schardl, 2010] propose a novel implementation of a worklist data structure, called bag, in place of FIFO queue usually employed in level-based algorithms. The use of this new structure and the experiments with different priority functions might promote more improvements to the algorithm studied in this work. Moreover, additional experiments on a machine with a higher number of available cores constitutes an important complementary test to be made in order to better explore the algorithm's scalability.

References

Budd, T. A. (2016). *An Active Learning Approach to Data Structures Using C.*

Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, p. 157–172, New York, NY, USA. ACM.

Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55. ISSN 1070-9924.

Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25. ISSN 0098-3500.

Farzaneh, A., Kheiri, H., and Shahmersi, M. A. (2009). An efficient storage format for large sparse matrices. *Communications Series A1 Mathematics & Statistics*, 58(2):1–10. ISSN 1303-5991.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA. ISBN 0716710455.

Hu, Y. F. and Scott, J. A. (2001). A multilevel algorithm for wavefront reduction. *SIAM Journal on Scientific Computing*, 23(4):1352–1375.

Karantasis, K. I., Lenharth, A., Nguyen, D., Garzarán, M., and Pingali, K. (2014). Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, p. 921–932, Piscataway, NJ, USA. IEEE Press. ISBN 978-1-4799-5500-8.

Karypis, G. and Kumar, V. (1998). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95. ISSN 0743-7315.

Kumfert, G. and Pothen, A. (1997). Two improved algorithms for envelope and wavefront reduction. *BIT Numerical Mathematics*, 37(3):559–590.

Leiserson, C. E. and Schardl, T. B. (2010). A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, p. 303–314, New York, NY, USA. ACM. ISBN 978-1-4503-0079-7.

Lin, W. (2005). Improving parallel ordering of sparse matrices using genetic algorithms. *Appl. Intell.*, 23(3):257–265.

Lin, Y. and Yuan, J. (1994). Profile minimization problem for matrices and graphs. *Acta Mathematicae Applicatae Sinica*, 10(1):107–112. ISSN 1618-3932.

Pooch, U. W. and Nieder, A. (1973). A survey of indexing techniques for sparse matrices. *ACM Comput. Surv.*, 5(2):109–133. ISSN 0360-0300.

Reid, J. K. and Scott, J. A. (2012). HSL package specification: MC60. Package version 1.1.0.

Rodrigues, T. N. (2017). tnas/reordering-library: XLIX Brazilian Symposium on Operational Research. URL `https://doi.org/10.5281/zenodo.438317`.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition. ISBN 0898715342.

Sánchez-Oro, J., Laguna, M., Duarte, A., and Mart, R. (2015). Scatter search for the profile minimization problem. *Networks*, 65(1):10–21.

Siek, J. G., Lee, L.-Q., and Lumsdaine, A. (2002). *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-72914-8.

Sloan, S. W. (1986). An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2):239–251.

Tewarson, R. P. (1973). *Sparse matrices*. Mathematics in science and engineering. Academic Press, New York.