Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200052

Improving the Scalability of the ABCD Solver with a Combination of New Load Balancing and Communication Minimization Techniques[1]

Iain DUFF [a,b], Philippe LELEUX [a,2], Daniel RUIZ [c], and F. Sukru TORUN [d]
[a] *CERFACS, Toulouse, France*
[b] *Scientific Computing Dpt., Rutherford Appleton Laboratory, Oxon, England*
[c] *IRIT - Institut de recherche en informatique de Toulouse, Toulouse, France*
[d] *Ankara Yildirim Beyazit University, Ankara, Turkey*

Abstract The hybrid scheme block row-projection method implemented in the ABCD Solver is designed for solving large sparse unsymmetric systems of equations on distributed memory parallel computers. The method implements a block Cimmino iterative scheme, accelerated with a stabilized block conjugate gradient algorithm. An augmented pseudo-direct variant has also been developed to overcome convergence issues. Both methods are included in the ABCD solver with a hybrid parallelization scheme. The parallel performance of the ABCD Solver is improved in the first non-beta release, version 1.0, which we present in this paper. Novel algorithms for the distribution of partitions to processes are introduced to minimize communication as well as to balance the workload. Furthermore, the master-slave approach on each subsystem is also improved in order to achieve higher scalability through run-time placement of processes. We illustrate the improved parallel scalability of the ABCD Solver on a distributed memory architecture by solving several problems from the SuiteSparse Matrix Collection.

Keywords. Block Cimmino, hybrid solver, sparse matrix, distributed memory parallelism, iterative solver

1. The iterative and augmented block-Cimmino method

The Augmented Block Cimmino Distributed Solver (ABCD Solver) is a distributed hybrid scheme designed to solve large sparse unsymmetric linear systems of the form:

[1]This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union. We acknowledge PRACE for awarding us access to MareNostrum at the Barcelona Supercomputing Center (BSC), Spain.
[2]Corresponding author: CERFACS, Toulouse, France; E-mail: leleux@cerfacs.fr

$$Ax = b, \tag{1}$$

where A is a full row rank $m \times n$ matrix, $m \leq n$, x is a vector of size n and b is a vector of size m. The approach is based on the *block Cimmino row projection method* (**BC**) [1]. BC is applied to the system which is partitioned in p row blocks where $p < m$. Starting from an arbitrary initial estimate $x^{(0)}$, a BC iteration improves the estimated solution by summing the projections of the current iterate on the subspaces spanned by the blocks of rows to converge to a solution. The convergence rate of BC is known to be slow [2]. When looking at the fixed point of the iterations, we obtain the following equivalent system:

$$Hx = k, \text{ where } \begin{cases} H = \sum_{i=1}^{p} \mathcal{P}_{\mathcal{R}(A_i^T)} = \sum_{i=1}^{p} A_i^+ A_i \\ k = \sum_{i=1}^{p} A_i^+ b_i. \end{cases} \tag{2}$$

As the row blocks A_i are assumed to have full row rank, H is symmetric and positive definite. To accelerate the convergence of the block Cimmino method, we solve instead this system using a block conjugate gradient algorithm (BCG) improved with stabilization of both residuals and directions [3]. The convergence of this method stays problem dependent and in some cases, convergence profiles with long plateaux can be observed. The eigenvalues of the matrix H are directly linked to the principal angles between subspaces spanned by the row partitions. If these angles are wider, the convergence becomes faster.

As an alternative, that we call **ABCD**, our solver also offers the possibility of constructing a larger system $\begin{bmatrix} A & C \\ B & S \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ f \end{bmatrix}$ where the numerical orthogonality between partitions is enforced. As a result, the block Cimmino method converges in exactly one iteration and x is the solution of the original system. This results in a pseudo-direct method [4] with the solution dependent on the projections as in BC, and on the direct solution of a system involving the matrix S. However, the efficiency of such an approach, compared to other sparse direct solvers, depends on the size or the density of the condensed system S which are problem dependent. Implementation of both ABCD and BC are available in the ABCD Solver[3] package.

2. Hybrid parallelism

In this section, we present the parallelization scheme of the ABCD Solver using MPI and OpenMP, and the need for an optimization of the load balancing and communication reduction. Both BC and ABCD methods perform the same preprocessing steps. Firstly, after scaling the system, we partition the matrix so that the principal angles between the subspaces given by the partitions are not too small, and the sizes of the partitions are balanced. There are many ways to construct these partitions. In the case of an iterative solution with BC, we will

[3] http://abcd.enseeiht.fr/

consider graph partitioners on the normal equations as they tend to reduce the number of iterations, as illustrated in [5]. In the case of the pseudo-direct solution with ABCD, we shall consider instead the multilevel hypergraph partitioner PaToH [6], which essentially decreases the size of the augmentation scheme, see [7]. Secondly, the basic idea is to distribute each partition to one process, called *master*, which builds an augmented system [8] used to compute the projection on the subspace spanned by the block of rows in the partition.

Thirdly, these augmented systems are solved using the sparse direct solver MUMPS[4] [9]. This direct solver uses the well-known multifrontal method and performs three steps: analysis (preprocessing, estimation of workload and memory), LDL^T factorization, and finally solve (forward elimination and backward substitution). Analysis and factorization must only be performed once, while one solve is needed to compute each projection at each iteration. These local projections are then summed through non-blocking point-to-point communications between masters. The amount of data communicated is equal to the number of shared columns, called *interconnections*. Note that additionally in ABCD, the matrix S is built in an embarrassingly parallel way by computing each column with a projection independently, then S is given in distributed form directly to MUMPS for a parallel solve on the global communicator (see [4] and [7] for the details of the construction and solution of S).

The ABCD Solver is a *hybrid scheme*, in the sense that the method is iterative but relies on a direct solver for each subproblem defined by the partition. The solver also implements a *hybrid parallelism* in the sense that several levels of parallelism are exploited at the same time:

1. the projections are independent and can be computed in parallel,
2. the MUMPS solver introduces two levels of parallelism: through the exploitation of its *elimination tree* and through the factorization of large frontal matrices using parallel linear algebra *dense kernels*.

Depending on the number of processes and the number of partitions, there are various possibilities for scheduling the computations. In the following sections of the article, we propose and study three different approaches for this. In the first approach, we consider an equal number of processes and partitions, in which case each master has exactly one partition. We experiment, in Section 3, to find the optimal number of processes per node to reduce the execution time.

In the second approach, the number of MPI processes is assumed to be less than the number of partitions. In such a case, the idea would then be to assign groups of partitions to the masters, which will construct one single block diagonal system, made with the various partitions. This block diagonal system can then be solved as before using MUMPS, and the goal is to balance the workload over all masters when distributing the partitions. In Section 4, we propose a new algorithm that aims to group partitions on each master so as to minimize the overhead of communication between masters, and at the same time equilibrate the load balance across masters.

In the third approach, we assume more MPI processes than partitions, in which case processes with no partitions can be associated with the masters, as

[4] http://mumps-solver.org/

slave processes, in order to contribute to the parallel computations in MUMPS. The target is to set master-slaves groups with balanced workloads, by taking into account the anticipated number of flops given by the MUMPS analysis of each partition. In Section 5, we first present a fast and optimal assignment of the slaves that balances the workload across subgroups of processes. Then we introduce a new method to assign the processes, masters and slaves, in the physical computing resources to decrease the communication overhead both within and between master-slaves groups depending on the method used, BC or ABCD.

In the ABCD Solver, we distinguish three types of communications [7]: the *inter-communication* between masters which occurs when summing the projections; the *intra-communication* inside master-slaves group which only occurs when computing a projection using MUMPS; finally in ABCD, *global communication* when solving the system based on S.

To illustrate the impact of our contributions, we run the ABCD Solver on three matrices from the SuiteSparse Matrix Collection [10]. Table 1 shows characteristics of the matrices. We conduct our experiments on MareNostrum4, a petascale supercomputer at the Barcelona Supercomputing Center[5]. It is a cluster with Intel Xeon Platinum processors. Each compute node is a 2-socket system where the 24 cores of each processor constitute a separate NUMA (non-uniform memory access) domain and nodes are interconnected with the Intel Omni-Path architecture. MareNostrum4 offers 96 GB RAM memory per NUMA domain, which means around 4 GB per core.

Table 1. Characteristics of the test matrices. n: the order of the matrix, nnz: the number of nonzero values in the matrix.

Matrix	n ($\times 10^6$)	nnz ($\times 10^6$)	nnz/n	kind
hamrle3	1.45	5.51	3.81	circuit simulation problem
cage15	5.15	99.20	19.24	directed weighted graph
memchip	2.70	13.00	4.93	circuit simulation problem

3. Optimal node configuration

When the number of partitions equals the number of processes, we determine the best distribution of MPI processes with respect to the execution time. With a fixed number of 128 MPI processes and an equal number of 128 partitions, we increase the number of processes per node from 2 to 64. Table 2 shows the execution times and we see that 2 MPI processes per node yields the minimum overall times. Although this results in more communication, because the linear algebra kernels used throughout the code and in MUMPS are memory-bound, they benefit from distributing the memory. Fewer processes per node implies less concurrent access to memory and faster computation. We will allocate 2 processes per node as our optimal configuration in the rest of the paper.
Since only a subset of nodes is used by MPI processes, when increasing the number of nodes we have the possibility of activating OpenMP parallelism but do not study this here where we focus on workload balancing and communication reduction.

[5]https://www.bsc.es/marenostrum/marenostrum

Table 2. Timings for the factorization of the augmented systems, for the BCG in BC, and for the pseudo-direct solution in ABCD. All runs were with 128 MPI processes spread with *ppn* processes per node and 128 partitions. Note that the memory required for ABCD was too large to solve the system cage15 on MareNostrum4.

Matrix	ppn	nodes	BC			ABCD	
			facto(s)	BCG(s)	it.	facto(s)	sol.(s)
Hamrle3	32	4	0.17	192	500	0.22	9.44
	16	8	0.19	138	”	0.21	9.48
	4	32	0.18	79	”	0.20	9.50
	2	64	0.18	77	”	0.22	10.10
cage15	32	4	1 550	65	17	-	-
	16	8	1 380	52	”	-	-
	4	32	1 230	40	”	-	-
	2	64	1 210	38	”	-	-
memchip	32	4	0.44	361	500	0.42	29.60
	16	8	0.31	269	”	0.31	29.70
	4	32	0.28	171	”	0.29	28.80
	2	64	0.27	168	”	0.29	28.10

4. Load balancing: distribution of partitions

In the case where the number of partitions is higher than the number of processes, a master process owns a group of partitions. In this section, the goal is to distribute the partitions to the masters with the right trade-off between balancing the weight of the local groups of partitions over all processes and minimizing the overhead in communication between masters.

4.1. Balancing the weight of the local partitions

We first consider only balancing the weights of the partitions. The weights should represent the future workload to compute projections. In the absence of more precise data at this point of the solver, we simply use the number of rows as a crude measure. Although this gives reasonable results here, it can result in bad load imbalance. In the next section, we will use accurately estimated workloads from a latter phase of the solver to distribute the slave processes. To balance the weights, we use the greedy algorithm introduced in [7]. The algorithm distributes partitions sorted in decreasing order of weights to masters. At each step, the master with current lowest accumulated weight receives a partition. This process results in an optimal distribution of the partitions over all masters in terms of balancing our criterion.

4.2. Minimize the overhead of communication

Globally, balancing the weights of local sets of partitions is not the only concern, one should also consider the overhead from inter-communication between masters resulting from the distributed sum of local projections and, in ABCD, from the parallel solution of the condensed system S. Therefore, the best distribution of the partitions should find the right trade-off between this communication, i.e. minimizing the number of interconnected columns between processors, and balancing the workload over processes in order to achieve minimum parallel execution time.

We propose a new algorithm which is based on this principle. The algorithm first creates a graph $\mathcal{G}$. The vertices of $\mathcal{G}$ are the partitions weighted by their respective size. There is an edge between two vertices if the corresponding partitions are interconnected, i.e. they share a nonzero column, and the cost of that edge equals the number of such columns. In the final step, we partition $\mathcal{G}$ using the multilevel graph partitioning tool METIS [11] to minimize the number of interconnections between the groups of partitions for each master, with a parameter μ that allows a certain imbalance in the accumulated weight over the groups of partitions.

4.3. Experimental results

The experiments are conducted on the three matrices with the greedy algorithm (*Greedy*) and the communication reducing algorithm where $\mu = 1\%$ (*Comm1*) and $\mu = 10\%$ (*Comm10*). Each matrix is partitioned into 1 024 blocks and is solved using 128 MPI processes spread over 64 distributed nodes with no multithreading. The numerically aware partitioning [5] is applied for BC, and the PaToH hypergraph partitioner is used for ABCD. Results are reported in Table 3. The column 'Com. col%' of Table 3 reports the total communication volume, equal to the number of interconnected columns, normalized with respect to the greedy method. The table also reports execution times for the factorization as well as the imbalance ratio between the slowest and average factorization times over all masters. Finally, the table gives the BCG execution time and iterations for BC, and the time to compute the pseudo-direct solution including the solution of the system S for ABCD.

As seen in the table, for BC, the proposed methods *Comm1* and *Comm10* achieve around 55% and 62% reduction in the total number of exchanged columns for the cases of Hamrle3 and memchip, respectively. This improvement in turn leads to faster parallel execution of BCG for Hamrle3 and memchip. Our experiments show that the larger ratio μ has a limited effect on the reduction of the total size of communication. On the other hand, for cage15, although there is considerable reduction in the communication values, the execution time increases slightly because the overhead of load imbalance absorbs the gain from the minimization of communication.

In the case of ABCD, there is only one iteration, thus each communication is only performed once. Compared to the gain of having balanced workloads over the MUMPS instances, the final communication overhead is low and thus the time only increases, slightly, with the proposed algorithm.

5. Placement of masters and slaves

In this section, we consider the case where there are more processes than partitions. We make use of the extra processes to act as slaves to help the master MPI processes to parallelize the computation further. We balance the workload over all masters by assigning more slaves to a master with a relatively higher workload.

Table 3. Impact of the distribution of partitions on the execution times. All runs were with 1024 partitions and 128 MPI processes on 64 nodes with no multithreading. (Com. col: Normalized column reduction values with respect to the Greedy algorithm. tot: Total time in seconds. it: Number of iterations required for convergence. imb: ratio of maximum over average factorization times. Sol. time: Total solution time in seconds)

Matrix	Algo.	BC					ABCD			
		Com.	Fact.		BCG		Com.	Fact.		Sol.
		col%	tot	imb	tot	it.	col%	tot	imb	time
Hamrle3	Greedy	100	0.22	1.62	714.25	4249	100	0.24	1.21	**8.17**
	Comm1	46	0.19	1.26	**700.66**	4249	42	0.25	1.35	9.12
	Comm10	45	0.20	1.36	713.69	4249	41	0.20	1.36	8.79
cage15	Greedy	100	20.41	1.97	**28.01**	18	-	-	-	-
	Comm1	44	42.61	3.94	34.62	18	-	-	-	-
	Comm10	44	48.82	3.75	35.00	18	-	-	-	-
memchip	Greedy	100	0.36	1.18	299.23	791	100	0.32	1.18	**5.64**
	Comm1	38	0.33	1.22	298.89	791	32	0.34	1.27	5.74
	Comm10	38	0.35	1.21	**292.05**	791	31	0.33	1.23	5.72

5.1. Assignment of the slaves

We consider w_k the accurate estimated workload of master $k \in \{1..nb\_masters\}$ given by MUMPS, i.e. the number of flops required for MUMPS factorization. We propose a new 2-step algorithm for the distribution of the slaves. Firstly, considering the number of slaves corresponding to the relative workload of each master k:
$s_k^{(theo)} = (w_k / \sum_{i=1}^{\#masters} w_i) \times \#slaves$, a number of slaves equal to the floor part of this amount is assigned to each master. Since most of the slaves are now associated with a master, the second step only has to allocate the remaining slaves. Secondly, we apply a greedy algorithm: at each step, one of the remaining slaves is assigned to the master-slave group with the currently highest average workload, until all slaves have been assigned. We obtain an optimal distribution of the slaves in terms of average workload and, thanks to the first step, the number of greedy searches performed is decreased.

5.2. Hierarchy of the computing architectures

The ABCD Solver is designed to solve large systems on distributed memory architectures where the computing resources are hierarchically structured, as is the case here with the supercomputer MareNostrum4.

When launching our distributed application, we specify a certain number of MPI processes per node which are allocated by the batch system. As a result, when the program starts, processes are already allocated and placed on the system architecture in a certain way. Depending on the situation at runtime, we need to decide which processes will be given the role of master or slave in order to minimize the total overhead of the communication between masters (inter-communication) on the one hand, and inside master-slaves groups (intra-communication) on the other hand. This process consists of three steps: firstly the placement of the masters, secondly the assignment of the number of slaves as in the last section using the estimation of the workload with MUMPS, and thirdly choosing the slaves for each master depending on its position in the architecture.

Two opposite approaches emerge in this situation. We can place masters close to each other to accelerate inter-communication, and we refer to this approach as *Compact*, or we place the master-slaves group together on a node to simultaneously improve intra-communication, and decrease concurrent access to memory by masters. We refer to this latter approach as *Scatter*.

5.3. Explicit placement of masters and slaves over nodes

The approach first implemented in the ABCD Solver, see [7], is Compact: the first ranks of MPI make the masters and the rest of the processes are assigned in a sequence to them as slaves depending on the rank. Although this approach minimizes the inter-communication, both the intra-communication as well as the sequential calls to dense kernels, known to be memory-bound, are slowed down due to concurrent memory access among masters.

Based on the results obtained in Section 3, mainly for BC, we have seen that spreading processes over the nodes is better because of more efficient memory access. Thus, we propose to implement the Scatter approach to improve the execution time of the ABCD Solver. Note that we currently use a "manual" implementation of this approach, but this could be replaced by architecture aware mechanisms in the future [12]. We define two algorithms for placement of the masters and the slaves.

The principle of these algorithms is simple:

- To place masters, we first gather information to know which node each process is on. We then assign one master per non-full node in a zig-zag fashion, starting from the biggest node to smallest then alternating.
- To place slaves, we first sort the masters in descending number of desired slaves. Then for each master, we place the slaves in the corresponding node and, if some are left, we group the remaining ones in other nodes as closely as possible.

In Figure 1, we illustrate the effect of the Compact and Scatter approach on a toy example. We partition a matrix in 3 partitions solved using block Cimmino with 12 processes. We define 3 masters each with 3 slaves and launch the solver on 3 nodes each with 4 processes.

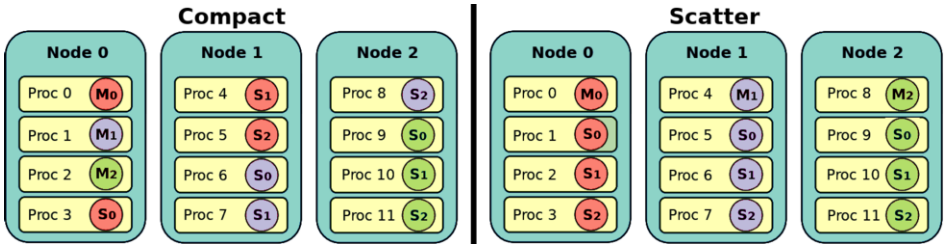

Figure 1. 3 nodes with 4 processes on each and we have 3 masters with 3 slaves each. M_i corresponds to the master i and the S_j of the same colour is its slave j. **(Left)** Compact scheme, **(Right)** Scatter scheme.

5.4. Experimental results

The results are presented in Table 4. Firstly, we observe that the execution times for factorization remain mostly unchanged for both algorithms for memchip and Hamrle3. In the case of cage15, which is dominated by this phase, the execution time of factorization is decreased in Scatter, benefiting from less concurrent access to memory. Concerning the BC method, the times for the sum of projections, which is included in the time for BCG, can increase in some cases with the Scatter approach, due to most master-slaves communicators being spread over the nodes. However, the overall BCG run-times always benefit from spreading the masters over the nodes, inducing less concurrency in memory access, and from grouping master-slaves groups, thanks to faster intra-communication. The effects of changing the algorithm are overall very small. In the end, we only have 2 processes per node so changing their placement does not change the global performance. We ran the experiment for the matrices Hamrle3 and memchip again, using 16 processes per node, thus 128 MPI processes on 8 nodes. The memory required for cage15 was too high for this configuration. Regarding the run-time of the BCG, Hamrle3 is solved in 203s with Compact and 161s with Scatter, while memchip is solved in 254s with Compact and 186s with Scatter. While the overall run-time with Scatter is higher than running with only 2 processes per node, the difference is only 4.5% for both matrices. Using the Compact algorithm however, the degradation is around 25%. This means that using the Scatter algorithm is more robust to having multiple active cores per node, which is a big step towards gaining scalability. However, in the case of ABCD the time to compute the pseudo-direct solution no longer benefits from spreading the masters with Scatter. In this approach, the computation is completely distributed, thus the overhead in communication absorbs the improvement from lower concurrent access to memory. Overall, the timings are not too different. Because of an implementation mixing together multiple layers of parallelism from MUMPS and the partitioning itself, the hybrid parallelism used is robust.

Table 4. Impact of the placement of masters and slaves on the execution times of ABCD Solver iterative method. All runs were with 32 partitions and 128 MPI on 64 nodes with no multithreading.

Matrix	Algo.	Block Cimmino				ABCD	
		facto(s)	BCG(s)	it.	proj. sum(s)	facto(s)	Sol.(s)
Hamrle3	Compact	0.41	159	500	76.4	0.36	43.2
	Scatter	0.40	154	500	77.4	0.33	45.1
cage15	Compact	567	22.7	15	14.6	-	-
	Scatter	560	22.3	15	14.7	-	-
memchip	Compact	0.40	184	365	89.1	0.46	24.8
	Scatter	0.43	178	365	85.8	0.45	28.8

6. Conclusion

We have shown the potential improvement that can be obtained in a master-slave scheme by considering the minimization of communication on an equal footing

with the balancing of workload. Firstly, we proposed a new distribution of partitions such that we decrease the communication between masters in the block Cimmino method, thus decreasing the total execution time in a context where many iterations are necessary with processes communicating for each iteration. Secondly, we propose a new way of attributing the roles of master or slave to processes depending on the run-time situation on the machine. We have identified two specific schemes : scattering the masters over the nodes is well adapted to the block Cimmino method, especially when the number of iterations is high, while compacting the masters in the same nodes is adapted for the augmented block Cimmino pseudo-direct method. Furthermore, the Scatter approach is more robust with respect to the number of processes per node, which is a big step towards scalability. Finally, we demonstrate the improved parallel scalability on a distributed memory architecture.

References

[1] Tommy Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[2] Randall Barry Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):168–193, 1992.

[3] Daniel Ruiz. Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment. *CERFACS TH/PA/9*, 6, 1992.

[4] Iain S Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block cimmino distributed method. *SIAM Journal on Scientific Computing*, 37(3):A1248–A1269, 2015.

[5] F. Torun, M. Manguoglu, and C. Aykanat. A novel partitioning method for accelerating the block cimmino algorithm. *SIAM Journal on Scientific Computing*, 40(6):C827–C850, 2018.

[6] Umit V Catalyürek and Cevdet Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[7] Mohamed Zenadi. *The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method*. PhD thesis, EDMITT, 2013.

[8] Åke Björck. Iterative refinement of linear least squares solutions i. *BIT Numerical Mathematics*, 7(4):257–278, 1967.

[9] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[11] G Karypis and V Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *Department of Computer Science, University of Minnesota*, 1995.

[12] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, 2013.