Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping*

François Pellegrini
Jean Roman
LaBRI, URA CNRS 1304
Université Bordeaux I
351, cours de la Libération, 33405 TALENCE, FRANCE
{pelegrin|roman}@labri.u-bordeaux.fr

Research Report 1038-96

Abstract

The combinatorial optimization problem of assigning the coexisting communicating processes of a parallel program onto a parallel machine so as to minimize its overall execution time is called static mapping. In this paper, we present a mapping algorithm based on the recursive bipartitioning of both the source process graph and the target architecture graph, whose divide and conquer and modular approach allows the handling of many topologies and bipartitioning methods. Specific experimental studies are carried out in order to validate the algorithm, and determine the conditions under which it achieves maximum efficiency. We analyze the interactions between the order in which the recursive bipartitionings are performed and the structure of the graphs to map; we evaluate the features of our implementation of the Fiduccia-Mattheyses algorithm for graph partitioning that allow it to handle weighted graphs; and we evidence the influence of the decomposition of the target topology on mapping quality. Then we outline the capabilities of Scotch 3.0, a software package for static mapping that implements this approach, and compare its performance to other mapping and partitioning software packages.

*This work was supported by the French GDR PRS

1 Introduction

The efficient execution of a parallel program on a parallel machine requires that the communicating processes of the program be assigned to the processors of the machine so as to minimize its overall running time. When processes are assumed to coexist simultaneously for the duration of all the program, this optimization problem is called *mapping*. It amounts to balancing the computational weight of the processes among the processors of the machine, while reducing the communication overhead induced by parallelism by keeping intensively intercommunicating processes on nearby processors. In many such programs, the underlying computational structure can be conveniently modeled as a graph in which vertices correspond to processes that handle distributed pieces of data, and edges reflect data dependencies. The mapping problem can then be addressed by assigning processor labels to the vertices of the graph, so that all processes assigned to some processor are loaded and run on it. In a SPMD context, this is equivalent to the *distribution* of data structures across processors; in this case, all pieces of data assigned to some processor are handled by a single process located on this processor.

A mapping is called *static* if it is computed prior to the execution of the program and is never modified at run-time. Static mapping is NP-complete in the general case [6]. Therefore, many studies have been carried out in order to find sub-optimal solutions in reasonable time. Specific algorithms have been proposed for mesh [18] and hypercube [4, 10] topologies. When the target machine is assumed to have a communication network in the shape of a complete graph, the static mapping problem turns into the *partitioning* problem, which has also been intensively studied [1, 13, 15, 23].

Scotch is a project carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I, by the ALiENor (ALgorithmics and ENvironments for parallel computing) team. Its goal is to study static mapping by the means of graph theory, using a "divide and conquer" approach. This work has resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and the analysis of several graph bipartitioning heuristics, all of which have been embodied in the Scotch software package for static mapping. This package allows the user to map efficiently any weighted source graph onto any weighted target graph, or even onto disconnected subgraphs of a given target graph, in an observed running time linear in the number of source edges and logarithmic in the number of target vertices.

This paper analyzes the results that have been obtained to date within the Scotch project. We introduce the DRB mapping algorithm, and determine the conditions under which it achieves maximal efficiency. We validate our design choices by means of experimental studies, which evidence the interactions between the structure of the algorithm and the topologies of the source and target graphs. The rest of the paper is organized as follows. Section 2 gives some definitions, and section 3 presents the most important aspects of the Dual Recursive Bipartitioning algorithm. The influence of the recursion process on the quality of the mappings is discussed in section 4, and section 5 deals with complexity issues. Sections 6 and 7 focus on the source and architecture bipartitioning algorithms that we use, respectively. In each of the four above sections, the solutions that we propose for the problems at stake are discussed, tested, and validated by means of experimental analyses; for the sake of clarity and readability, these are described within the bodies of the sections, rather than in a separate part of the paper. Section 8 describes the capabilities of the SCOTCH 3.0 software package for static mapping, and section 9 compares its performance to other static mapping and partitioning software packages such as CHACO [12] and METIS [16]. Then follows the conclusion.

2 Definitions

2.1 Static mapping

The parallel program to be mapped onto the target architecture is modeled by a weighted unoriented graph S called *source graph* or *process graph*. Vertices v_S and edges e_S of S are assigned integer weights $w(v_S)$ and $w(e_S)$, which estimate the computation weight of the corresponding process and the amount of communication to be transmitted on the inter-process channel, respectively. The target machine onto which is mapped the parallel program is also modeled by a valuated unoriented graph T called *target graph* or *architecture graph*. Vertices v_T and edges e_T of T are assigned integer weights $w(v_T)$ and $w(e_T)$, which estimate the computational power of the corresponding processor and the cost of traversal of the inter-processor link, respectively. A *mapping* of a source graph S onto a target graph T consists of two applications $\tau_{S,T} : V(S) \longrightarrow V(T)$ and $\rho_{S,T} : E(S) \longrightarrow \mathcal{P}(E(T))$, where $\mathcal{P}(E(T))$ denotes the set of all the simple loopless paths that can be built from $E(T)$. $\tau_{S,T}(v_S) = v_T$ if process v_S of S is mapped onto processor v_T of T, and $\rho_{S,T}(e_S) = \{e_T^1, e_T^2, \ldots, e_T^n\}$ if communication channel e_S of S is routed through communication links e_T^1, e_T^2, $\ldots$, e_T^n of T. $|\rho_{S,T}(e_S)|$ denotes the dilation of edge e_S, that is the number of edges of $E(T)$ used to route e_S.

2.2 Cost functions

The computation of efficient static mappings requires an *a-priori* knowledge of the dynamic behavior of the target machine with respect to the programs which are run on it. This knowledge is synthesized in a *cost function*, the nature of which determines the characteristics of the desired optimal mappings. Cost functions may account for criteria such as the load balance on the target processors, the communication load balance on the communication links, the minimization of inter-processor communication, the minimization of the dilation of the edges of the source graph, *etc.* In general, several such criteria are combined into a unique aggregate cost function by means of weighted sums. However, the biggest drawback of aggregate functions lies in the setting of the weighting coefficients. In particular, the trade-off between computation and communication criteria is hard to tune, and must be evaluated for every different target machine.

Therefore, as several authors did before [5, 17, 23], we have chosen to separate computation criteria from communication ones. The goal of our mapping algorithm is thus to minimize some communication cost function, while keeping the load balance within a user-specified tolerance. The communication cost function f_C that we have chosen is the sum, for all edges, of their dilation multiplied by their weight:

$$f_C(\tau_{S,T}, \rho_{S,T}) \stackrel{\text{def}}{=} \sum_{e_S \in E(S)} w(e_S)\, |\rho_{S,T}(e_S)| \ .$$

This function, which has already been considered by several authors for hypercube target topologies [4, 10, 12], has several interesting properties: it is easy to compute, allows incremental updates performed by iterative algorithms, and its minimization favors the mapping of intensively intercommunicating processes onto nearby processors; regardless of the type of routing implemented on the target machine (store-and-forward or cut-through), it models the traffic on the interconnection network and thus the risk of congestion. The strong positive correlation between its values and effective execution times has been experimentally verified by several authors [10, 13].

2.3 Performance criteria

The quality of mappings is evaluated with respect to the criteria for quality that we have chosen: the balance of the computation load across processors, and the minimization of the interprocessor communication cost modeled by function f_C. These criteria lead to the definition of several parameters, whose expressions are given below.

For load balance, one can define μ_{map}, the average load per computational power unit (which does not depend on mappings), and δ_{map}, the load imbalance ratio, as

$$\mu_{map} \stackrel{\text{def}}{=} \frac{\sum\limits_{v_S \in V(S)} w(v_S)}{\sum\limits_{v_T \in V(T)} w(v_T)} \qquad \text{and}$$

$$\delta_{map} \stackrel{\mathrm{def}}{=} \frac{\displaystyle\sum_{v_T \in V(T)} \left| \left(\frac{1}{w(v_T)} \sum_{\substack{v_S \in V(S) \\ \tau_{S,T}(v_S) = v_T}} w(v_S) \right) - \mu_{map} \right|}{\displaystyle\sum_{v_S \in V(S)} w(v_S)} .$$

However, since the maximum load imbalance ratio is provided by the user in the input of the mapping, the information given by these parameters is of little interest; what matters is the minimization of the communication cost function under this load balance constraint.

For communication, the salient parameter to consider is f_C. It can be normalized as μ_{exp}, the average edge expansion, which can be compared to μ_{dil}, the average edge dilation; these are defined as

$$\mu_{exp} \stackrel{\mathrm{def}}{=} \frac{f_C}{\displaystyle\sum_{e_S \in E(S)} w(e_S)} \qquad \text{and} \qquad \mu_{dil} \stackrel{\mathrm{def}}{=} \frac{\displaystyle\sum_{e_S \in E(S)} |\rho_{S,T}(e_S)|}{|E(S)|} .$$

$\delta_{exp} \stackrel{\mathrm{def}}{=} \frac{\mu_{exp}}{\mu_{dil}}$ is smaller than 1 when the mapper succeeds in putting heavily intercommunicating processes closer to each other than it does for lightly communicating processes; it is equal to 1 if all edges have same weight.

A mapping will be said better than another if its communication cost f_C is smaller than the one of the other graph, and provided that its load imbalance is below the user-defined tolerance.

2.4 Test graphs

The source graphs that have been used to test our mapping program belong to two distinct classes. The first one is made of triangular and quadrangular unstructured meshes related to fluid dynamics, structural mechanics, or combinatorial optimization problems. The computations performed for every vertex of these graphs being supposed identical, they are all homogeneous, that is have unity vertex and edge weights. The second class contains valuated interprocess communication graphs issued from a parallel implementation of a sparse block Cholesky factorization solver, which represent partitions of the unknowns induced by a nested dissection method. The most imbalanced graph, *REF0*, is the direct output of the nested dissection process. Others are obtained from this one by means of a refinement process, in which heavier vertices are split into cliques of lighter vertices. This allows for better granularity of the problem, at the expense of vertex and –mostly– edge creations. Therefore, most of these graphs have very high degree; see [3] for reference.

The characteristics of all these graphs are summed-up in table 1. δ and Δ stand for the minimum and maximum degrees of the graphs, respectively. In all this paper, *diam* denotes graph diameter, and M_2, H, and K represent the bidimensional grid, hypercube, and complete graphs, respectively.

Name	Class	$\|V(S)\|$	$\|E(S)\|$	δ	Δ	min $w(v_S)$	max $w(v_S)$	min $w(e_S)$	max $w(e_S)$
3ELT	2D F.E.	4720	13722	3	9	1	1	1	1
4ELT	2D F.E.	15606	45878	3	10	1	1	1	1
4ELT2	2D F.E.	11143	32818	3	12	1	1	1	1
BODY	3D F.E.	45087	163734	0	28	1	1	1	1
BUMP	2D F.E.	9800	28989	3	8	1	1	1	1
BCSSTK29	3D F.E.	13992	302748	4	70	1	1	1	1
BCSSTK30	3D F.E.	28924	1007284	3	218	1	1	1	1
BCSSTK31	3D F.E.	35588	572914	1	188	1	1	1	1
BCSSTK32	3D F.E.	44609	985046	1	215	1	1	1	1
BRACKET	3D F.E.	62631	366559	3	32	1	1	1	1
OCEAN	3D F.E.	143437	409593	1	6	1	1	1	1
PWT	3D F.E.	36519	144794	0	15	1	1	1	1
ROTOR	3D F.E.	99617	662431	5	125	1	1	1	1
SPHERE	3D F.E.	16386	49152	4	6	1	1	1	1
REF0	N.D.	2047	7750	2	186	167	686298	6	4560
REF1	N.D.	2453	47659	2	444	167	270419	1	1403
REF2	N.D.	2815	84406	2	542	167	83652	1	1104
REF3	N.D.	3093	105713	2	584	167	38749	1	444
REF4	N.D.	3470	135148	2	633	167	25910	1	264

Table 1: Characteristics of the source graphs used for our tests.

2.5 Experimental conditions

Our tests have been carried out on a SGI Onyx machine with 190 MHz R10000 processors and 128 Mb of main memory. The measured times are total CPU times (user and system) taken to compute mappings, excluding data loading and results saving. Except if explicitly mentioned, all the mapping computations used logarithmic indexing, adaptive sequencing, and the `gfx` strategy, with a load imbalance tolerance ratio of 0.005; these parameters will be defined in the following, when needed.

3 The Dual Recursive Bipartitioning algorithm

3.1 Description of the algorithm

Our mapping algorithm uses a *divide and conquer* approach to recursively allocate subsets of processes to subsets of processors [19]. It starts by considering a set of processors, also called the *domain*, containing all the processors of the target machine, and with which is associated the set of all the processes to map. At each step, the algorithm bipartitions a yet unprocessed domain into two disjoint subdomains, and calls a *graph bipartitioning algorithm* to split the subset of processes associated with the domain across the two subdomains. Whenever a domain is restricted to a single processor, its associated processes are assigned to it and recursion stops, as written in the following sketch.

```
mapping (D, P)
Set_Of_Processors  D;
Set_Of_Processes   P;
{
  Set_Of_Processors  D0, D1;
  Set_Of_Processes   P0, P1;

  if (|P| == 0)          /* If nothing to do. */
    return;
  if (|D| == 1) {        /* If one processor in D */
    result (D, P);       /* P is mapped onto it.  */
    return;
  }

  (D0, D1) = processor_bipartition (D);
  (P0, P1) = process_bipartition   (P, D0, D1);
  mapping (D0, P0);      /* Perform recursion. */
  mapping (D1, P1);
}
```

The association of a subdomain with every process defines a *partial mapping* of the process graph. The *complete mapping* is achieved when successive bipartitionings have reduced all subdomain sizes to one.

The above algorithm lies on the ability to define five main objects:

- a *domain structure*, which represents a set of processors in the target architecture.
- a *domain bipartitioning function*, which, given a domain, bipartitions it in two disjoint subdomains.
- a *domain distance function*, which gives, in the target graph, a measure of the distance between two disjoint domains. Since domains may not be convex nor connected, this distance may be estimated. However, it must respect certain homogeneity properties, such as giving more accurate results as domain sizes decrease. The domain distance function is used by the graph bipartitioning algorithms to compute the communication function to minimize, since it allows the mapper to estimate the dilation of the edges that link vertices which belong to different domains. Using such a distance function amounts to considering that all routings use shortest paths on the target architecture. This is not unreasonable to assume, as most existing parallel machines handle routing dynamically with shortest-path routings. We have thus chosen that our program would not provide routings for the communication channels, leaving their handling to the communication system of the target machine.

- A *process subgraph structure*, which represents the subgraph induced by a subset of the vertex set of the original source graph.

- A *process subgraph bipartitioning function*, which bipartitions subgraphs in two disjoint pieces to be mapped onto the two subdomains computed by the domain bipartitioning function.

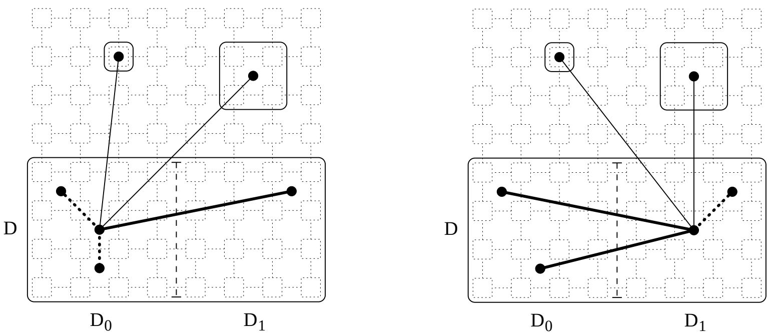
All of these routines are seen as black-boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions (see sections 6 and 7).

3.2 Partial cost function

The production of efficient complete mappings requires that all graph bipartitionings favor the criteria that we have chosen. Therefore, the bipartitioning of a subgraph S' of S should maintain load balance within the user-specified tolerance, and minimize the *partial* communication cost function f'_C, defined as

$$f'_C(\tau_{S,T}, \rho_{S,T}) \stackrel{\text{def}}{=} \sum_{\substack{v \,\in\, V(S') \\ \{v,v'\} \,\in\, E(S)}} w(\{v,v'\})\,|\rho_{S,T}(\{v,v'\})| \ ,$$

which accounts for the dilation of edges internal to subgraph S' as well as for the one of edges which belong to the cocycle of S', as shown in figure 1. Taking into account the results of partial mappings issued by previous bipartitionings makes it possible to avoid local choices that might prove globally bad, as explained below.



a. Initial position. **b.** After one vertex is moved.

Figure 1: Edges accounted for in the partial communication cost function when bipartitioning the subgraph associated with domain D between the two subdomains D_0 and D_1 of D. Dotted edges are of dilation zero, their two ends being mapped onto the same subdomain. Thin edges are cocycle edges.

4 Job sequencing schemes

4.1 Sequencing schemes

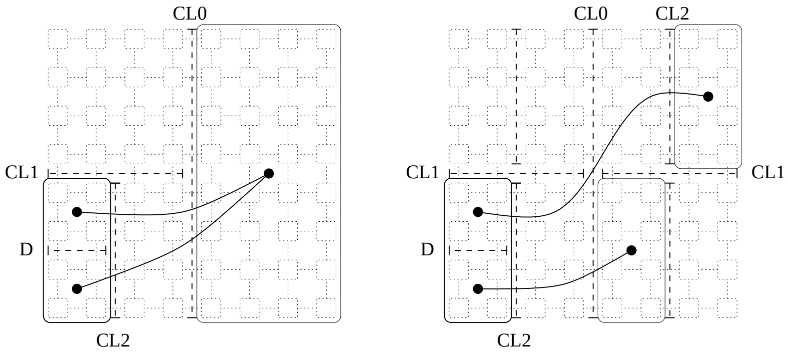
From an algorithmic point of view, our mapper behaves as a greedy algorithm, since the assignment of a process to a subdomain is never reconsidered. The double recursive call performed at each step induces a recursion scheme in the shape of a binary tree, each vertex of which corresponds to a bipartitioning job, that is the bipartitioning of both a domain and its associated process subgraph.

In the case of depth-first sequencing, as written in the above sketch, bipartitioning jobs run in the left branches of the tree have no information on the distance between the vertices they handle and neighbor vertices to be processed in the right branches. On the contrary, sequencing the jobs according to a by-level (breadth-first) travel of the tree allows that any bipartitioning job of a given level may have information on the subdomains to which all the processes have been assigned at the previous level. Thus, when deciding in which subdomain to put a given process, a bipartitioning job can account for the communication costs induced by all the neighboring processes, whether they are handled by the job itself or not, since it can estimate in f'_C the dilation of the corresponding edges. This results in an interesting feed-back effect: once an edge has been kept in a cut between two subdomains, the distance between its end vertices will be accounted for in the partial communication cost function to be minimized, and following jobs will be more likely to keep these vertices close to each other, as illustrated in figure 2. Moreover, since all domains are split at every level, they all have equivalent sizes, which respects the distance homogeneity and gives the algorithm more coherence (see section 7.3).

We have defined a third sequencing scheme, called *adaptive*, which selects the job which has the highest number of cocycle edges linking it to jobs that handle subgraphs with fewer vertices than it has. The goal of this adaptive scheme is to emulate breadth-first sequencing, while using as much as possible mapping results produced by jobs of the same levels. As a matter of fact, no job will be selected if bigger jobs (that is, jobs belonging to higher levels) remain unselected, and once a job has been selected in a level, jobs which share edges with it will be selected next, so that they can use the most accurate distance information regarding these cocycle edges.

4.2 Evaluation of the sequencing schemes

Figure 3 illustrates the advantages of adaptive sequencing compared to breadth-first sequencing. Figure 3.a represents the *BUMP* mesh. Figure 3.b shows the partial result of the first two levels of the mapping of this graph onto the bidimensional grid $\mathrm{M}_2(4,2)$: with every graph vertex is associated a disk whose grey level indicates the subdomain to which the vertex currently belongs; two vertices have disks of same grey level if they are mapped onto the same subdomain.



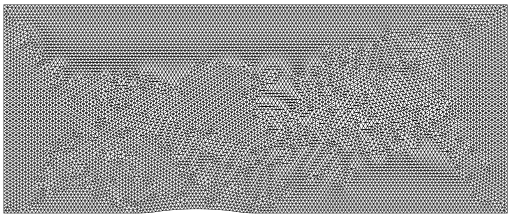
a. Depth-first sequencing. **b.** Breadth-first sequencing.

Figure 2: Influence of depth-first and breadth-first sequencings on the bipartitioning of a domain D belonging to the leftmost branch of the bipartitioning tree. With breadth-first sequencing, the partial mapping results regarding vertices that belong to the right branches of the bipartitioning tree are more accurate (C.L. stands for “Cut Level”).

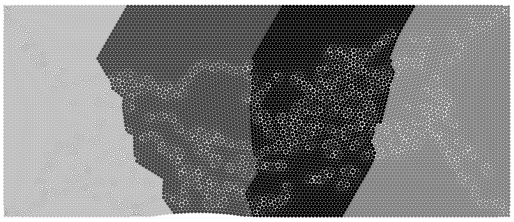
At this point, breadth-first and adaptive sequencings give the same result, since the recursive decomposition of the graph into bands of same orientation zeroes the contribution of cocycle edges, such that the order in which jobs are bipartitioned in the first two levels is irrelevant. However, this is no longer true at the next level.

Figure 3.c shows the final mapping of *BUMP* onto $\mathrm{M}_2(4,2)$ using breadth-first sequencing. Let the four subdomains of figure 3.b be labeled “1”, “2”, “3”, and “4” from left to right. With breadth-first sequencing, they are bipartitioned in order “3”, “4”, “1”, “2”. Subdomain 3 is the first to be bipartitioned, and therefore has no useful cocycle information to use. Then subdomain 4 is bipartitioned, and takes advantage of the result of the bipartitioning of subdomain 3 to minimize the number of cocycle edges of dilation 2: the frontier of the resulting bipartition matches the one computed for subdomain 3. However, when subdomain 1 is bipartitioned, it has no information on how subdomain 2 is going to be bipartitioned. Therefore, it assigns at random the two resulting subgraphs to the two processors it handles. When subdomain 2 is finally bipartitioned, it must account for the allocation performed for subdomains 1 and 3. Unfortunately, the two processors of subdomain 1 have been associated with the subgraphs generated by the bipartition in a way opposite to the one of subdomains 3 and 4, which results in a “twisted” mesh. To accommodate for this, the job which bipartitions subdomain 2 builds an overlapping “diagonal” bipartition which tends to minimize the local communication cost function by reducing as much as possible the number of edges of dilation 2.

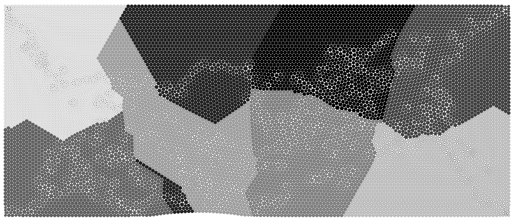
On the other hand, with adaptive sequencing, bipartitioning jobs are selected



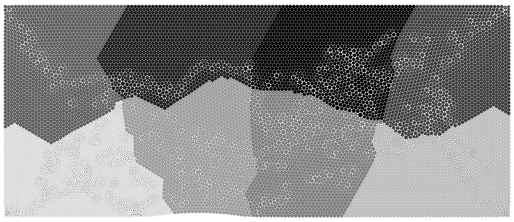
a. The *BUMP* mesh.



b. Result of the first two levels of the mapping of *BUMP* onto $\mathrm{M}_2(4,2)$.



c. Result of the mapping of *BUMP* onto $\mathrm{M}_2(4,2)$ with the breadth-first sequencing.



d. Result of the mapping of *BUMP* onto $\mathrm{M}_2(4,2)$ with the adaptive sequencing.

Figure 3: Result of the mapping of graph *BUMP* onto $\mathrm{M}_2(4,2)$ with breadth-first and adaptive job sequencings.

provided that at least one of their neighbors has been processed. Therefore, once one of the four subdomains has been bipartitioned, processors are associated with subgraphs so that the mesh is not twisted, and frontiers can match across subdomains. This propagation of mapping results between neighboring jobs yields the more regular and efficient mapping of figure 3.d.

To evaluate the respective efficiency of these three sequencing schemes, we have used all of them to map our test graphs onto hypercubes and square bidimensional grids of increasing sizes. Globally, adaptive sequencing is the most efficient scheme in term of average expansion, providing the best mappings of the three in 48.2 percent of the runs, followed by depth-first sequencing with 39.9 percent, and then breadth-first sequencing with 35.1 percent (the sum is over 100 percent since several schemes can give the same best result). In fact, breadth-first sequencing does better than depth-first sequencing on average. However, when this happens, it is most often outperformed by adaptive sequencing, so from now we will only compare adaptive and depth-first sequencings. When only these two schemes are taken into account, the adaptive scheme computes the best mappings in 63.7 percent of the runs, and the depth-first scheme in 51.2 percent of the runs. However, results differ significantly according to the target topology and the type of source graph, as shown in table 2. Note that, in this table, we have summed-up the results over columns and rows. Although aggregating values obtained for graphs of different nature may not seem correct, we did it to show that, regardless of one of the parameters, the other has a significant impact on mapping results.

	Source									
	N.D.		2D F.E.		3D F.E.		all F.E.		all	
Target	a.s.	d.f.s.	a.s.	d.f.s.	a.s.	d.f.s.	a.s.	d.f.s.	a.s.	d.f.s.
$\mathrm{H}(x)$	57.7	53.3	55.5	55.5	74.1	40.7	63.5	49.2	61.1	50.9
$\mathrm{M}_2(x,x)$	44.0	72.0	80.0	40.0	93.3	33.3	85.7	37.1	68.3	51.6
all	52.9	60.0	64.3	50.0	81.0	38.1	71.4	44.9	63.7	51.2

Table 2: Percentage of finding the best mapping for the adaptive (a.s.) and depth-first (d.f.s.) sequencings, for several classes of source graphs and target architectures.

For nested-dissection graphs, depth-first sequencing is more efficient on average than adaptive sequencing; it does better for large hypercubes, and its superiority is obvious for bidimensional grids, with 72.0 percent against 44.0 percent. Because of the high density and heavy edge weights of nested-dissection graphs, knowing more accurately many edge dilations (and particularly the ones of the heaviest edges) compensates the risk of computing worse partial mappings in the left branches of the bipartitioning tree, all the more when edge dilations may be large, as it is the case for grid target architectures.

For finite-element meshes, adaptive sequencing clearly outperforms depth-first sequencing. When source graphs are loosely connected, exhibit great locality, and are of small dimensionality, the feed-back and of propagation effects that we have discussed above help to "unfold" the source graph on the target architecture as efficiently as possible. The depth-first scheme behaves better for hypercube topologies since the high degree and small diameter of these graphs limit the consequences of bad choices made in the left branches of the sequencing tree, while providing the jobs of the right branches accurate distance information regarding the vertices which have been already mapped.

The conclusion of this study is that the efficiency of job sequencing schemes strongly depends on the nature of the source and target graphs. When source graphs are strongly connected and/or have heavily weighted edges, depth-first sequencing does better than the breadth-first-like adaptive sequencing, because the contribution of the heaviest edges dominates the cost function. On the opposite, when source graphs are loosely connected, exhibit great locality, and are of small dimensionality, adaptive sequencing is much more efficient in preserving this locality in the resulting mapping. Adaptive sequencing should therefore be preferred when mapping finite-element meshes onto parallel architectures.

As a closing remark, one can note that, by using the hypercube as target topology and depth-first sequencing, our mapping algorithm becomes very similar to the one of Ercal, Ramanujam, and Sadayappan [4]. In that sense, our work, by formalizing the concepts of domain, distance, and execution scheme, can be seen as a generalization of theirs that handles any target topology and graph bipartitioning method.

5 Complexity analysis

The purpose of this section is to evaluate the complexity of our recursive mapping algorithm with respect to the ones of the bipartitioning methods that are used within the bipartitioning jobs.

Let ALGO be an algorithm. We note $\mathcal{C}_{Spa}(ALGO)$ the maximal space complexity of this algorithm. It represents the biggest amount of memory which the algorithm may need during its execution, this memory space being freed at completion. Similarly, we note $\mathcal{C}_{Tim}(ALGO)$ the maximal time complexity of the algorithm. The interest of some heuristics is that their effective behavior can be several orders of magnitude below their maximal complexity, although this cannot be mathematically proven. Therefore, we note $\mathcal{C}'_{Spa}(ALGO)$ the space behavior of algorithm ALGO, and $\mathcal{C}'_{Tim}(ALGO)$ its time behavior. Unlike complexity, which is a theoretical, proven, result, the behavior of an algorithm is an empirical result, only based on experimentation. However, it is reliable, because observed in a quasi systematic way on a great number of examples, and justified by qualitative arguments.

Let BipaT and BipaS be the domain and subgraph bipartitioning algorithms used by our DRB algorithm.

Proposition 1 *Let S be a source graph and T be a target graph, with $|E(S)| \geq |E(T)|$, $|V(S)| \geq |V(T)|$, and $|E(S)| \geq |V(S)|$. If, for all $T' \subseteq T$ and $S' \subseteq S$, $\mathcal{C}_{Spa}(BipaT(T'))$ is in $O(|E(T')| + |V(T')|)$ and $\mathcal{C}_{Spa}(BipaS(S'))$ is in $O(|E(S')| + |V(S')|)$, and if BipaT gives subdomains of equivalent sizes, then $\mathcal{C}_{Spa}(DRB(S,T))$ is in $O(|E(S)|)$.*

Proof. By using neighbor lists, S is stored in $O(|V(S)| + |E(S)|)$ space. Since, by hypothesis, BipaT gives subdomains of equivalent sizes, the domain bipartitioning binary tree is complete at least up to its before-last level, and its depth is $\lceil \log_2(|V(T)|) \rceil$. Let $\mathcal{F}_i(S,T)$, with $i \geq 0$, be the set of (T',S') pairs processed by the bipartitioning jobs at level i of the bipartitioning tree, where T' is a subdomain of T and S' the subgraph of S mapped onto T'. In particular, $\mathcal{F}_0(S,T) = \{(S,T)\}$.
Since bipartitioning jobs free after completion the memory space that they use, the space complexity of our DRB algorithm is the maximum over all jobs of their space complexity:

$$\mathcal{C}_{Spa}(DRB(S,T)) = O(|E(S)|) + O(|V(S)|) + O\left(\max_{i=0}^{\lceil \log_2(|V(T)|) \rceil} \max_{\substack{(S',T') \in \\ \mathcal{F}_i(S,T)}} \max(\mathcal{C}_{Spa}(BipaT(T')), \mathcal{C}_{Spa}(BipaS(S'))) \right) .$$

At each level of the bipartitioning tree, all process subgraphs and subdomains are disjoint subgraphs of the initial source and target graphs, which are therefore of smaller sizes. The above expression is thus clearly bounded by the complexity of the first level, so

$$\mathcal{C}_{Spa}(DRB(S,T)) = O(|E(S)|) .$$

□

In the same way, the time complexity of our DRB algorithm can be computed under the same conditions.

Proposition 2 *Let S be a source graph and T a target graph, with $|E(S)| \geq |E(T)|$, $|V(S)| \geq |V(T)|$, and $|E(S)| \geq |V(S)|$. If, for all $T' \subseteq T$ and $S' \subseteq S$, $\mathcal{C}_{Tim}(BipaT(T'))$ is in $O(|E(T')| + |V(T')|)$ and $\mathcal{C}_{Tim}(BipaS(S'))$ is in $O(|E(S')| + |V(S')|)$, and if BipaT gives subdomains of equivalent sizes, then $\mathcal{C}_{Tim}(DRB(S,T))$ is in $O(|E(S)| \log_2(|V(T)|))$.*

Proof. Using the definitions of the previous proposition, we have

$$\mathcal{C}_{Tim}(DRB(S,T)) = O\left(\sum_{i=0}^{\lceil \log_2(|V(T)|)\rceil} \sum_{(S',T')\in\mathcal{F}_i(S,T)} (\mathcal{C}_{Tim}(BipaT(T'))+ \mathcal{C}_{Tim}(BipaS(S')))\right) .$$

At each level of the bipartitioning tree, all subdomains and subgraphs are disjoint subgraphs of the initial source and target graphs. Therefore, the sum for each level of complexities linear in the number of vertices and edges of these subgraphs is at most linear in the number of vertices and edges of the initial source and target graphs. Thus,

$$\begin{aligned}\mathcal{C}_{Tim}(DRB(S,T)) &= O\left(\sum_{i=0}^{\lceil \log_2(|V(T)|)\rceil} O(|E(T)|+|V(T)|+|E(S)|+|V(S)|)\right)\\ &= O\left(|E(S)|\lceil \log_2(|V(T)|)\rceil\right) .\end{aligned}$$

□

For what precedes, provided that the time behaviors $\mathcal{C}'_{Tim}$ of all our bipartitioning methods are in $O(|E(S')|+|V(S')|)$, we should have $\mathcal{C}'_{Tim}(DRB(S,T)) = O(|E(S)| \log_2(|V(T)|))$. Thanks to the bipartitioning methods that we use, this proves true, and is verified by two sets of experiments. The first one deals with the linearity in the number of edges of source graphs. We have mapped source graphs onto many target topologies, and obtain in all cases plots that have a linear shape (see for instance figure 4). The second set of experiments confirms that the running time of our mapper is logarithmic in the number of vertices of the target graph as long as the number of processors is smaller than the number of processes. We have mapped source graphs onto many target topologies of the same family, and obtain in all cases plots which have a logarithmic shape (and therefore which are of linear shape with a log/lin plotting, as in figure 5).

6 Process graph bipartitioning methods

The core of our recursive mapping algorithm uses process graph bipartitioning methods as black boxes. It allows the mapper to run any type of graph bipartitioning method compatible with our criteria for quality. Bipartitioning jobs maintain an internal image of the current bipartition, which indicates for every vertex of the job whether it is currently assigned to the first or the second subdomain. It is therefore possible to apply several different methods in sequence, each one starting from the result of the previous one, and to select the methods with respect to the job characteristics, which permits us to define *mapping strategies*.

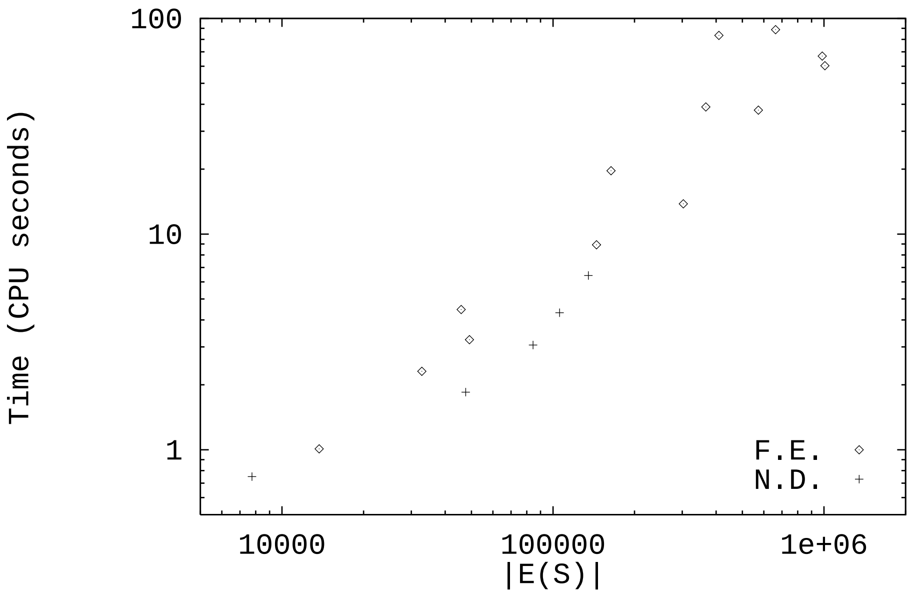


Figure 4: Running time of the mapping of finite-element (F.E.) and nested-dissection (N.D.) graphs onto the bidimensional grid $M_2(8,8)$, with strategy `gfx`.

6.1 Graph bipartitioning methods

Several graph bipartitioning methods have been implemented to date: random and greedy algorithms to compute initial bipartitions and refine them, a backtracking method, and an improved version of the Fiduccia-Mattheyses heuristic.

6.1.1 The Gibbs-Poole-Stockmeyer method

This bipartitioning method is inspired from an algorithm proposed by Gibbs, Poole, and Stockmeyer to minimize the dilation of graph orderings, that is the maximum absolute value of the difference between the numbers of neighbor vertices [8]. The graph is sliced by using a breadth-first spanning tree rooted at a randomly chosen vertex, and this process is iterated by selecting a new root vertex within the last layer as long as the number of layers increases. Then, starting from the current root vertex, vertices are assigned layer after layer to the first subdomain, until half of the total weight has been processed. Remaining vertices are then allocated to the second subdomain.

As for the original Gibbs, Poole, and Stockmeyer algorithm, it is assumed

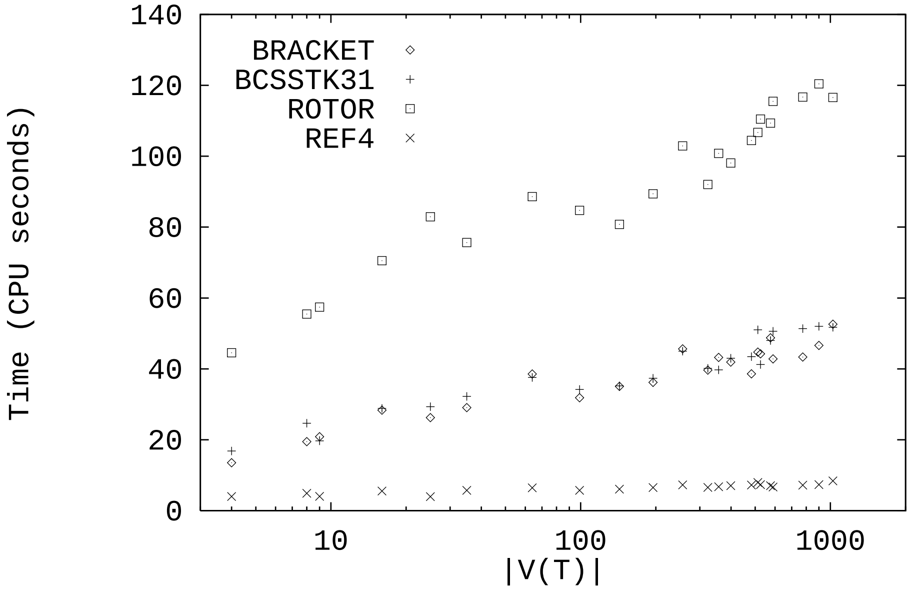


Figure 5: Running time of the mapping of some finite-element and nested-dissection graphs onto bidimensional meshes $M_2(x, y)$ of increasing dimensions, with strategy **gfx**.

that the maximization of the number of layers results in the minimization of the sizes –and therefore of the cocycles– of the layers. This property has already been used by George and Liu to reorder sparse linear systems using the nested dissection method [7], and by Simon in [24].

6.1.2 The Improved Fiduccia-Mattheyses method

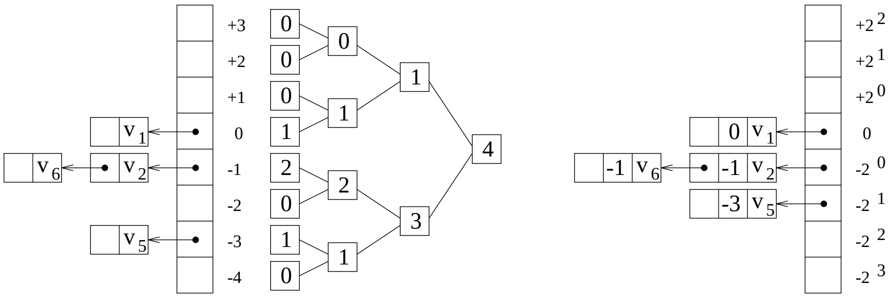
The Fiduccia-Mattheyses graph bipartitioning heuristic [5] is an almost-linear improvement of the Kernighan-Lin algorithm [17]. Its goal is to minimize the cut between two vertex subsets, while maintaining the balance of their cardinals within a limited user-specified tolerance. Starting from an initially balanced solution of any cut value, it proceeds iteratively by trying, at each stage, to reduce the cut of the current solution. The algorithm maintains, for all the vertices of both subsets, a *gain* value, which represents the value by which the current cut would decrease if the vertex were moved to the other subset (gains may thus be negative). Vertices of identical gain are linked into *gain linked lists* which are stored as entries of a *gain array*. At each stage, the algorithm builds an *order-*

ing of as many vertices as it can, by repeating the following process: it picks in the two subsets the vertex of largest current gain and whose move will not set the balance out of the user-specified tolerance. When a vertex is selected, the algorithm fakes to move it and updates the gains of its neighboring vertices accordingly. This is repeated until all vertices have been chosen or no move of yet unprocessed vertices would keep the load balance within the tolerance. Once the ordering is complete, the new solution is built from the current one by moving as many vertices of the ordering as necessary to get the maximum accumulated gain. Thus, by considering the accumulated gain, the algorithm allows *hill-climbing* from local minima of the cut cost function.

The almost-linearity of this algorithm is based on two limitations. First, it is supposed that vertex degrees are small and that all edges have unity weight, so that the range of the gain values is small, and thus the search in the gain array for a vertex of best gain is assumed to take an almost-constant time. Second, all vertices are supposed to have equal (unity) weights, so that moving the head of a given gain list is equivalent in balance to moving any vertex in this list. Unfortunately, the above is no longer true when vertices and edges have non-unity weights and when gains account for the distance between edge ends. As a matter of fact, the handling of huge weights raises three problems.

The first one is the access time to the linked list of largest gain. To solve it, we have first added to each gain array a binary-tree structure that recorded the number of vertices present in every sub-tree (see figure 6.a). Thanks to this, the search for the linked list of largest gain was performed in a time logarithmic in the size of the gain array; it was thus bounded by the number of address bits of the machine which ran the mapper, and therefore considered as constant. However, the above solution amplified the second problem, which is space consumption.

Since the biggest gain value which can be obtained during a mapping can be as large as $\mathrm{diam}(T)\,\Delta(S)\max_{e_S\in E(S)} w(e_S)$, the size of linear gain arrays can be prohibitive for valuated graphs. To keep the gain table a reasonable size guaranteeing an almost-constant access time, we have finally implemented a logarithmic indexing of gain tables, as shown in figure 6.b. In this case, the number of entries of the gain array is bounded by the number of bits coding an integer, which is a small constant. The differences in behavior between the algorithms with linearly and logarithmically indexed gain arrays are small, since in both cases vertices with big gains are handled first. As a matter of fact, when vertices linked within the same logarithmically indexed gain list are not neighbors, moving one of them does not modify the gain of the others, and the order of their moving has no importance, whatever indexing type is considered. If they are neighbors, the choice of a neighbor rather than another can influence the behavior of the algorithm; however, the resulting approximation is of the same order of magnitude as the one inherent to the initial Fiduccia-Mattheyses algorithm. Moreover, when the initial bipartition is already well structured (as



a. Linearly indexed gain array, flanked by a binary-tree structure summing list cardinalities. The traversal of the highest non-empty branches leads to the non-empty list of biggest gain.

b. Logarithmically indexed gain array. A gain list with strictly positive index i holds the vertices whose gains are between $\lfloor 2^{i-2} + 1 \rfloor$ and 2^{i-1}.

Figure 6: These two data structures both allow for the searching of a vertex of big(gest) gain in constant time. However, the second structure is much more efficient in space than the first one.

for instance when the Gibbs-Poole-Stockmeyer algorithm is used to compute an initial partition), the area within which vertices are swapped is small, and the risk for it to move is small.

The third problem caused by weighted graphs regards load balance. Because of the non-unity vertex weights, moving the head of a gain list may not result in the best load balance compared to other vertices in the same list, and moving a weighted vertex from the heaviest subdomain to the other can even increase load imbalance instead of reducing it, as for instance when moving the vertex with weight 3 in the imbalanced assignment of figure 7. As no data structure could easily avoid the evaluation of vertices which would cause imbalance if they were moved, and since we did not want to scan the whole lists for the vertex of best load balance, we have chosen to scan the gain list of best logarithmic gain till a vertex whose move would not cause imbalance is found. This is not unreasonable to do, since the imbalance tolerance defines a valid working space which should be fully utilized in order to jump over local minima of the communication cost function.

To ensure the validity of logarithmic indexing and study the consequences of grouping vertices of different gains into the same gain lists, we have replaced the entries of the logarithmic gain table by sub-arrays indexed by the leftmost non-zero bits of the gain values. Therefore, vertices that have the same leftmost non-zero bit, and which would have been stored in the same list in the logarithmic gain table, may now be stored in the distinct gain lists of the sub-array, according to the value of their leftmost non-zero bits, that we call sub-bits. Table 3

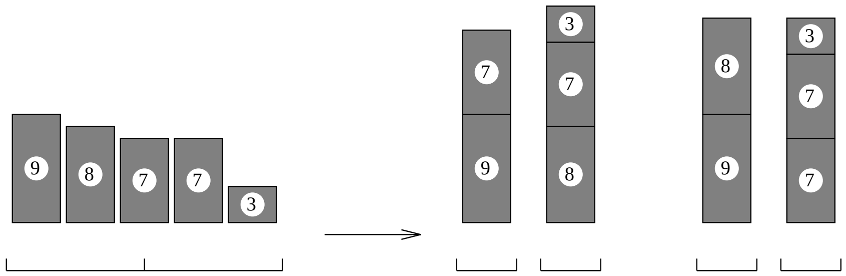


Figure 7: Imbalanced assignment computed by one of our greedy bipartitioning algorithms, and optimal assignment.

summarizes the average expansions of the mapping with the **gfx** strategy (see section 6.3) of graph *ROTOR* onto square bidimensional grids, using linear and logarithmic indexings of the gain tables of the Fiduccia-Mattheyses method; for logarithmic indexing, several numbers of sub-bits have been tested. As expected, the linear and logarithmic indexings give equivalent results, and none of them proves significantly better than the other. One can note that, in this example, the expansion values for linear and logarithmic indexing are equal for a number of sub-bits greater than or equal to 4. It means that, for these mappings, using 4 sub-bits generates enough distinct gain list entries to store all the different gain values used in the bipartitioning process, which results in a behavior identical to the one of linear indexing. However, logarithmic indexing proves more efficient in practice, since for instance graph *REF4* could not be mapped onto $\mathrm{M}_2(32, 32)$ with linear indexing, because of gain overflow, in spite of the allocation of a linear gain array with 2^{19} entries.

$\|V(T)\|$	Linear	Logarithmic (number of sub-bits)			
		0	1	2	4
4	0.00880	0.01199	0.00883	0.00883	0.00880
16	0.04666	0.05037	0.04493	0.04509	0.04666
64	0.12422	0.13114	0.12144	0.12352	0.12422
256	0.29499	0.31046	0.28697	0.29175	0.29499
1024	0.65186	0.68044	0.63095	0.64586	0.65186

Table 3: Average expansion of the mapping of graph *ROTOR* onto square bidimensional grids, with the **gfx** strategy, and using linear and logarithmic (with 0, 1, 2, and 4 sub-bits) indexings of the gain tables of the Fiduccia-Mattheyses method.

6.1.3 The "exactifying" algorithm

This method is used to post-process other methods when strict load balance is essential. The goal of this greedy algorithm is to reduce load imbalance while keeping the value of the communication cost function as small as possible. The vertex set is scanned in order of decreasing vertex weights, and vertices are moved from one subdomain to the other if doing so reduces load imbalance. When several vertices have same weight, they are inserted in a gain table analogous to the one used by our Fiduccia-Mattheyses algorithm, so that the vertex whose swap decreases most the communication cost function is selected first.

6.2 Handling of imbalanced graphs

Imbalanced process graphs, that is graphs such that some processes have weights much heavier than the average, lead to several problems. Let us consider for instance a complete process graph such that all vertices have weight one, except for one single vertex whose weight is equal to the number of vertices of the graph. If an exact bipartitioning algorithm were run on this graph, the heaviest vertex would be assigned to one subdomain and all the others vertices to the other, leaving all processors of the first subdomain idle, but one, as shown in figure 8. Imbalanced process graphs may be handled in two different ways, according to the two opposite following arguments.

- On one hand, one could think that since the heaviest process will complete last, it is useless to spread the other processes over all the processors. It is better to balance the load over a smaller number of processors, and leave the rest idle so that they can be allocated to other users of the parallel machine.
- On the other hand, since the processors of the domain have already been reserved, it is better to use them all, and spread processes as much as possible. Processes will therefore use the independent resources of every processor (computational power, memory, local disks, ...) with smaller risk of contention.

We have chosen to implement the second approach, since it allows to take advantage of all the resources that have been reserved for the execution of the parallel program, while allowing features similar to the ones of the first approach if we choose to restrict the target domain before the mapping process takes place. As a matter of fact, let S be the source graph to be mapped onto the target architecture T that has been reserved for execution. If we want all working processors to be loaded at least as much as the one that will host the heaviest

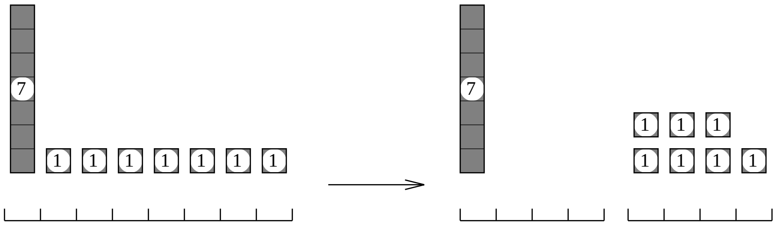
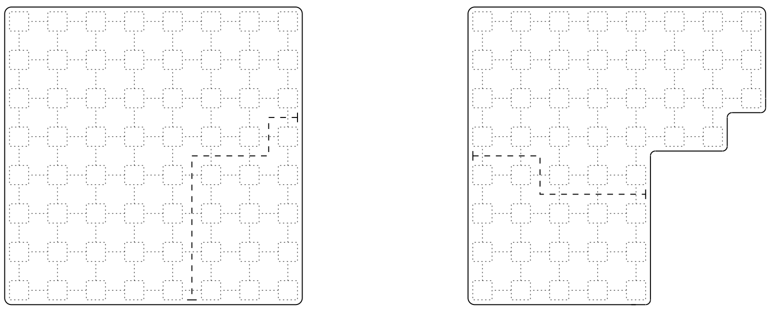


Figure 8: Exact bipartitioning of a strongly imbalanced graph.

process, the target architecture must be restricted to $T' \subseteq T$ such that

$$|V(T')| = \min\left(|V(T)|, \left\lfloor \frac{\sum\limits_{v_S \in V(S)} (w(v_S))}{\max\limits_{v_S \in V(S)} (w(v_S))} \right\rfloor\right) .$$

The construction of T' from T can be performed by our mapper, by mapping T onto the weighted path graph with two vertices of weights equal to the desired $|V(T')|$ and to $|V(T)| - |V(T')|$, respectively. This unique weighted bipartitioning tends to minimize the cut between the two resulting subgraphs of T, and therefore to maximize communication within each of them. T' is taken as the subgraph of T that has been mapped onto the vertex of weight $|V(T')|$, as illustrated in figure 9.



a. Construction of a subdomain with 13 vertices (lower right corner) on a 8 by 8 bidimensional grid architecture.

b. Construction of a subdomain with 17 vertices (lower left corner) on the remaining architecture.

Figure 9: Subdomain construction on $\mathrm{M}_2(8,8)$ by weighted bipartitioning.

According to the second approach that we have chosen, and in order to prevent excessive weights from perturbating the bipartitioning process, we have implemented an adaptive weight limitation procedure. At the beginning of every job, it computes the *effective weights* of all the processes, which are the minimum of the *real weights* of the processes and a multiple of the mean real weight of the processes assigned to the current domain. The multiplicative factor is dynamically adjusted so that the mean of the effective weights be centered with respect to all the effective weights. The use of effective weights by the bipartitioning algorithms amounts, for imbalanced subgraphs, to computing load balance more in term of numbers of processes than in term of process loads, which reduces the impact of pathological cases such as the one described above.

6.3 Evaluation of the partitioning methods

As said before, the different bipartitioning methods that we have implemented can be combined into what we call *strategies*. In this study, we have focused on four strategies: `f`, the plain Fiduccia-Mattheyses method; `fx`, which uses the exactifying method in post-processing; `gf`, which feeds the Fiduccia-Mattheyses method with the result of the Gibbs-Poole-Stockmeyer method; and `gfx`, which post-processes the latter with the exactifier. In order to compare these four strategies, we have used them to map our test graphs onto the hypercube, bidimensional grid, and complete graph topologies. Mapping results significantly differ according to the structure of the source graphs. We present below qualitative results, which we illustrate by experiments carried out on the hypercube target topology.

For nested dissection graphs (see figures 10 and 11), the use of the `g` method always improves mapping quality. The strong hierarchical structure of these graphs is very well suited for their clustering into layers, which yields very good initial partitions. On the opposite, post-processing the partitions with the `x` method always increases the cost function. Since these graphs are of high degree and have large edge weights, the `f` method tends to minimize the cost function by unbalancing the partitions so as to prevent heavy edges from being kept in the cut. Enforcing strict balance is therefore likely to turn these heavy edges into cut edges, and thus to increase the communication cost. Consequently, the exactifier should not be used for weighted graphs if cut minimization and connectivity are essential.

For finite-element meshes (see figures 12 and 13), the methods used in pre- and post-processing of the Fiduccia-Mattheyses method have only little impact on the quality of the mappings. This is due to the topological properties of triangular meshes of low dimensionality, which are fairly well suited for the Fiduccia-Mattheyses heuristic. Unlike what we have observed for nested dissection graphs, the use of the `x` method does not increase the average expansion. Since these graphs are not weighted, the exactifier moves the vertices that penalizes less the cost function. Most often, these are vertices which have some neighbors in the other subdomain, so that the exactifying process only results in a small displacement of the cut, which does not reflects upon the quality of

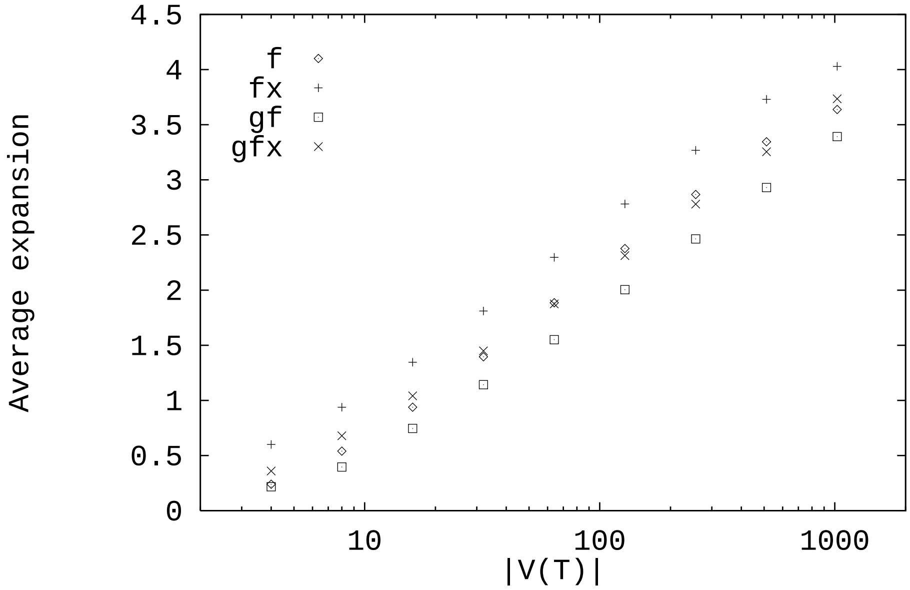


Figure 10: Average expansion of the mapping of graph *REF4* onto H(x) with several strategies.

the bipartitionings. A most useful result regards execution times: the computation of initial bipartitions by means of the **g** method, if it does not improve the quality of the partitions in this case, decreases computation times by 20 to 30 percent on average, by reducing the number of passes necessary for the Fiduccia-Mattheyses algorithm to converge. This is not new, as several authors insisted on the importance of providing good starting partitions to avoid premature termination due to local minima of the cost function, and to achieve faster convergence [9, 15]. From the above, the most suited strategy to map unweighted finite-element meshes is the **gfx** strategy, which is used in all the experiments of section 9.

7 Domain bipartitioning methods

According to the type of target architecture, the values of the domain functions can be algorithmically computed at run-time, or be extracted from precomputed look-up decomposition tables.

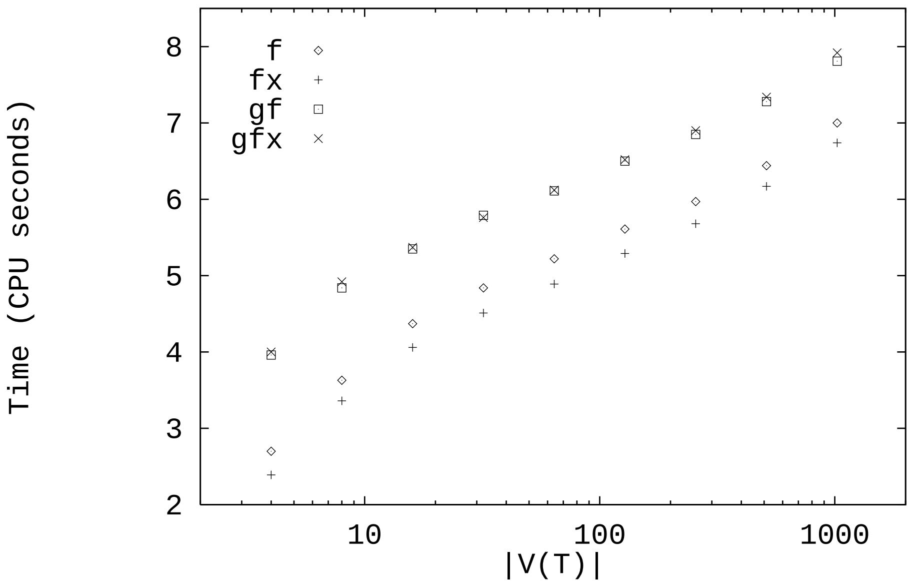


Figure 11: Running time of the mapping of graph *REF4* onto H(x) with several strategies.

7.1 Domain bipartitioning by mapping

Since, in our approach, the recursive bipartitionings of target graphs are fully independent with respect to the ones of source graphs (yet, the opposite is false), the recursive decomposition of a given target architecture needs only to be computed once, in order to store the resulting data in look-up decomposition tables which will be used in the mapping process. Decomposition tables can be easily computed with our mapper, by mapping the considered target graph onto the complete graph with same number of vertices. Mapping onto the complete graph zeroes the contribution of cocycle edges (since every subdomain is at distance 1 from all the others), so that only local cut minimization is considered. In the resulting decomposition, strongly-connected clusters of processors will be kept uncut as long as possible, and strongly-connected clusters of processes will therefore tend to be mapped onto them. In the case of heterogeneous architectures, the minimization of the communication function favors the cut of the edges with smallest weights, that is of biggest bandwidth. From the communication point of view, we obtain a hierarchical decomposition in which links of highest bandwidth act as backbones between subdomains containing

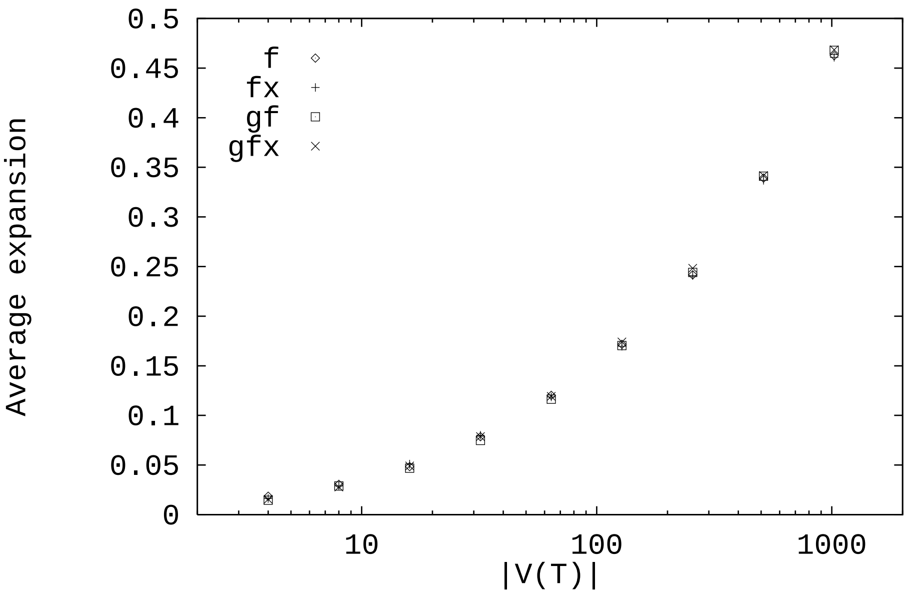


Figure 12: Average expansion of the mapping of graph *BRACKET* onto H(x) with several strategies.

links with smaller bandwidth.

The mapping of a source graph onto any target graph can be performed in two steps. First, the mapping engine is called a first time to map the target graph onto the complete graph, in order to build the decomposition table for this architecture. Then, it is called a second time to map the source graph onto the target architecture, using the decomposition table that has just been computed.

7.2 Algorithmic domain bipartitioning

The algorithmic handling of some specific architectures (meshes, hypercubes, complete graphs, multi-stage networks...), which may seem redundant with respect to the general-purpose decomposition table mechanism, allows the mapper to handle huge regular target architectures without storing tables whose sizes evolve as the square of the number of processors.

For instance, the bidimensional grid target architecture is algorithmically implemented such that domains are rectangular areas, the domain bipartitioning function splits a domain along its smallest dimension into two parts of equiv-

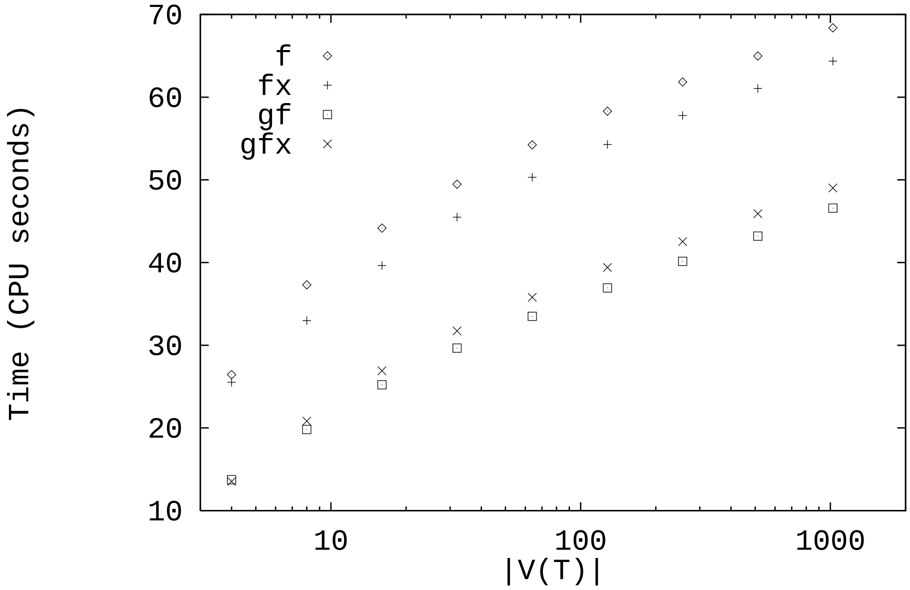


Figure 13: Running time of the mapping of graph *BRACKET* onto H(x) with several strategies.

alent sizes (within one row or column), and the distance function returns the distance between the centers of the two domains. For hypercube target architectures, domains are sub-hypercubes, the domain bipartitioning function splits a hypercube into two sub-hypercubes, and the domain distance function returns the number of bits which differ in the labeling of sub-hypercubes.

However, for non-standard target architectures (such as, for instance, a non-rectangular subdomain of a bidimensional grid architecture), the built-in functions cannot be used, and a proper decomposition table must be computed.

7.3 Evaluation of the decomposition techniques

In order to compare the two above decomposition techniques, we have mapped our test graphs onto several target architectures, using decompositions computed either algorithmically or by mapping onto the complete graph.

The target architecture that we have studied most is the bidimensional grid, since the members of this family may have any non-power-of-two number of vertices, and their decompositions can be represented and analyzed easily. The algorithmic decomposition of bidimensional meshes that we have implemented in our mapper uses a nested dissection approach such that square domains are always partitioned along the same dimension, which gives very regular decomposition patterns, as in figure 14. On the opposite, the cuts of square domains computed by mapping may not all have the same orientation (they may not even be straight), since only local cut minimization is achieved and thus no global coherence can be maintained, as illustrated in figure 15. The same holds for the hypercube topology as well: all the bipartitionings computed at the same level of an algorithmic decomposition have the same orientation, which may not be the case for mapped decompositions.

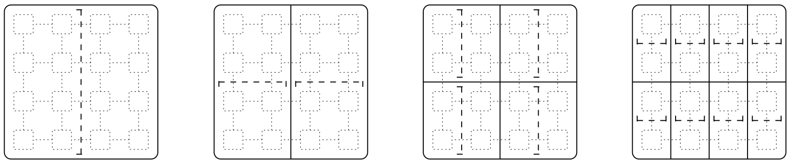


Figure 14: Algorithmic decomposition of the $M_2(4,4)$ architecture.

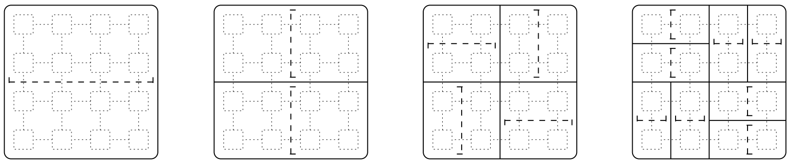


Figure 15: Decomposition of the $M_2(4,4)$ architecture by strictly balanced mapping onto $K(16)$. Unlike for algorithmic decompositions, the orientation of the cuts of square domains may vary.

Figure 16 shows the average expansion of the mapping of graph *BRACKET* onto bidimensional grids of various sizes that have been decomposed algorithmically, by mapping with strict enforcement of load balance (that is, by using our exactifying algorithm in post-processing), and by mapping with a weak load balance constraint of 10 percent. In all our experiments on bidimensional grids, the expansions computed using strictly mapped decompositions are from 5 to 10 percent above the ones of algorithmic decompositions, except for a few cases for which interactions between the behavior of the Fiduccia-Mattheyses algorithm, the shape of the graph, and the structure of the decomposition, lead to better results. This relative inefficiency of strictly mapped grid decompositions has two main reasons.

- The first one is the difficulty for the Fiduccia-Mattheyses graph bipartitioning method to handle bidimensional grids as source graphs. As a matter of fact, let us consider the smallest connected subset of the vertices of a graph that are assigned to the same subdomain and such that this subset is stable, that is such that the removal of a vertex from the subset does not improve the cut. Among all the regular uniform meshes, the bidimensional grid is the topology that allows stable subsets of smallest size (see figure 17). Consequently, for this architecture, stable subsets are easier to obtain than for other types of meshes, which increases the probability of the algorithm to be trapped in local minima of the communication cost function. This behavior had already been diagnosed by Gilbert and Zmijewski [9] for the Kernighan-Lin heuristic.

- The second reason comes from the strict enforcement of load balance. When a grid of odd dimensions is bipartitioned for the first time, the frontier between the two resulting subdomains exhibits a step-shaped discontinuity (see figure 18). As bipartitionings go on, this phenomenon can lead to the creation of "L"-shaped subdomains. To bipartition such a subdomain while minimizing the cut, the algorithm assigns to one of the subdomains the curved section of the "L", and to the other its two ends. This results in the construction of disconnected subdomains, which perturbates the computation of the distance function since this do not respect the principle of locality that we have introduced in section 3. The disconnected bipartitioning of "L"-shaped domains is favored by the 4-connectivity of the grid, for which cutting along a diagonal is twice as expensive as cutting along a dimension.

The impact of this second phenomenon can be reduced by performing imbalanced bipartitionings instead of strictly balanced bipartitionings. For instance, using an imbalance ratio of 10 percent decreases the average expansion of the mappings in more than 80 percent of the runs, as illustrated in figure 16.

For the hypercube target topology, the overcost induced by strictly mapped decompositions is of 2 to 3 percent over algorithmic ones, as shown for graph *BRACKET* in figure 19. These ratios are smaller than for the bidimensional grid because of the comparatively large degree and small diameter of the hypercube topology, which limit the impact of bad mappings. Moreover, hypercubes always have number of vertices which are powers of two, as well as a very regular structure, which always induce with the Fiduccia-Mattheyses algorithm a regular bipartitioning along some dimension. One can besides note that the smallest stable set of $H(n)$ is $H\left(\left\lceil \frac{n}{2} \right\rceil\right)$, since swapping any vertex that has less than $\left\lceil \frac{n}{2} \right\rceil$ neighbors in its own subdomain reduces the cut. Increasing the imbalance tolerance of the bipartitionings has no effect for hypercubes, since the minimum cut is achieved by strictly balanced bipartitioning in some dimension [2]. However, this may result in a cut along a dimension other than the one that would have been selected in the case of strict bipartitioning, and therefore may yield slightly different mapping results.

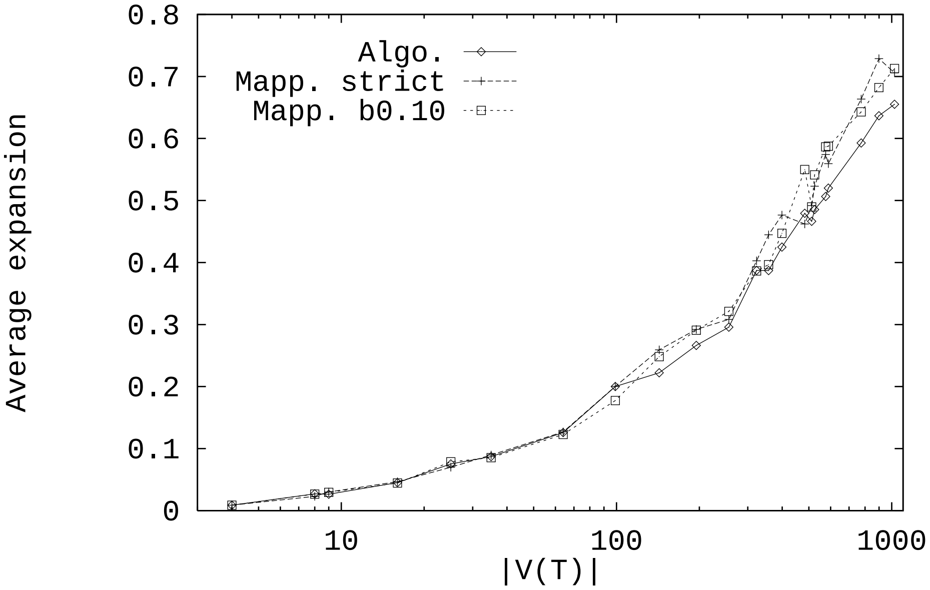


Figure 16: Average expansion of the mapping of graph *BRACKET* onto bidimensional grids $\mathrm{M}_2(x, y)$ decomposed algorithmically, by strictly balanced mapping, and by mapping with a balance constraint of 10 percent.

The above experiments on the bidimensional grid tend to prove that cut minimization is more important than load balance, since enforcing strictly balanced bipartitionings can lead to decompositions which perturbate the behavior of the distance function. In order to understand the mutual influences of the structure of the decompositions on the behavior of the distance function, we have mapped our test graphs onto algorithmic decompositions of bidimensional grids that have been computed by three different methods: the plain (within one row or one column) nested dissection algorithm presented above, an imbalanced nested dissection algorithm which allocates one third of the vertices to a subdomain and the remaining two thirds to the other, and a multiple one-way dissection algorithm that performs all recursive bipartitionings along some dimension before considering other dimensions.

The resulting expansions for graph *BRACKET* are given in figure 20. In almost all the cases, balanced nested dissection behaves best, and multiple one-way dissection worst. The unefficiency of multiple one-way dissection is due to the impact of the decomposition on the behavior of the distance function. As a matter of fact, the principle of the Dual Recursive Bipartitioning algorithm is to

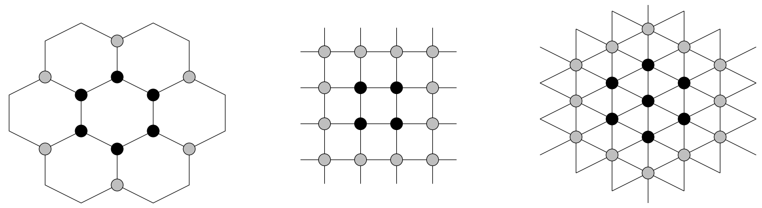


Figure 17: Smallest stable subsets for regular uniform mesh graphs. Moving one single vertex from the black part to the grey one does not improve the communication cost function, that is does not decrease the number of edges whose ends are of different colors.

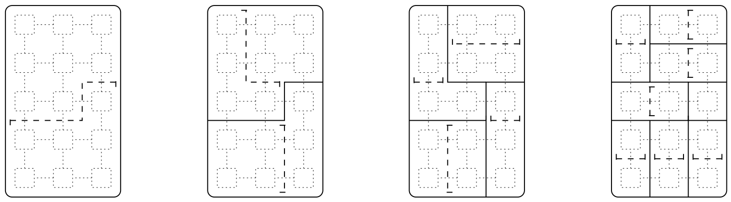


Figure 18: Decomposition of the $\mathrm{M}_2(3,5)$ architecture by strictly balanced mapping onto $\mathrm{K}(15)$.

refine the partial mapping induced by recursive bipartitionings of domains and their associated process sets, up to give a complete mapping when all subdomains are of size one. The efficiency of the algorithm is a result of the feed-back effect that we have discussed in section 4: every bipartitioning job uses the values of the current partial mapping to evaluate its local communication cost function, and optimize accordingly the bipartition that it computes. In order for the distance function to be accurate and reliable, so that the decisions made at some level do not prove bad at the next, its variations must decrease as domain sizes diminish. However, for multiple one-way dissection, the variability of the distance between domains, which decreases as bipartitionings are performed in the first dimension, roughly increases when the first bipartitioning is performed in the second dimension, as shown in figure 21. Therefore, this bipartitioning and the following ones, which have a greater impact on the cost function than the preceding ones, are performed after them, when the number of degrees of freedom of the problem has already been significantly reduced; therefore, they may not optimize the bipartition as they would have done if they had been considered before. This is why nested-dissection schemes (whether balanced or imbalanced), which respect the property of the distance function and perform costly bipartitionings when the number of degrees of freedom is maximal, yield better results. This is also why the balanced scheme is the most efficient: in the unbalanced scheme, bipartitionings that deal with large subdomains may

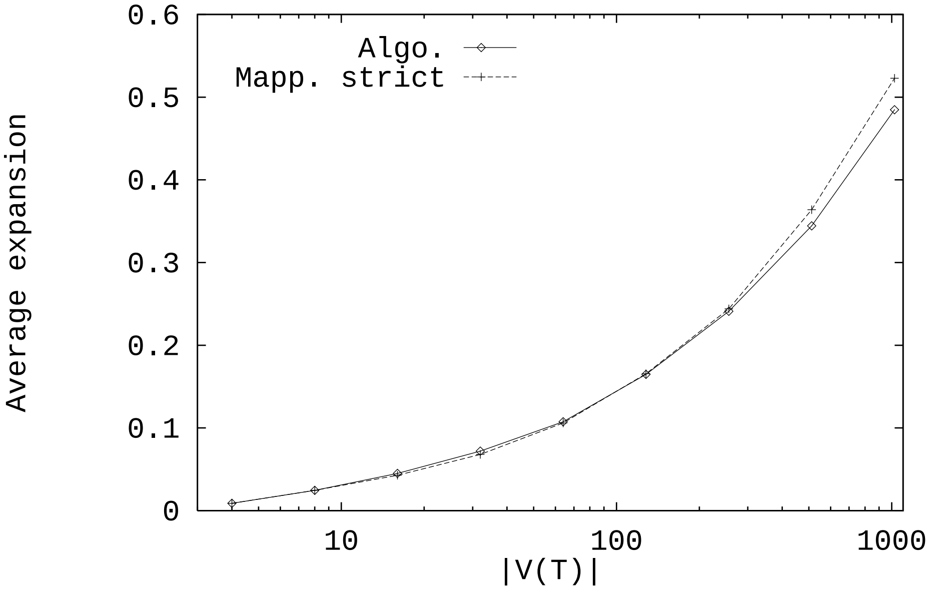


Figure 19: Average expansion of the mapping of graph $BRACKET$ onto hypercubes H(x) decomposed algorithmically and by strictly balanced mapping.

be carried out after ones that deal with smaller subdomains.

The tests carried out in this section show that algorithmic decompositions based on algorithmically computed nested dissections yield in almost all cases mappings of better quality than the ones that use decompositions computed by mapping. This is not really surprising, since the definition of decomposition algorithms requires some knowledge of the topological properties of the considered target architectures, which can be exploited to provide more regular and efficient decompositions that preserve the properties of the distance function.

As a matter of fact, a most important result of these experiments is the formalization of the characteristics of the distance function that make it suitable for the Dual Recursive Bipartitioning algorithm. To produce mappings of quality, the distance function must be such as to give more accurate results as the sizes of the end domains diminish, and such that its variations decrease accordingly. These properties are coherent with the local nature of the DRB algorithm, which tries to make the less informative choices at first, and refines the partial mappings as domain sizes diminish. In practice, we have observed that target architectures that do not allow to define decompositions that respect

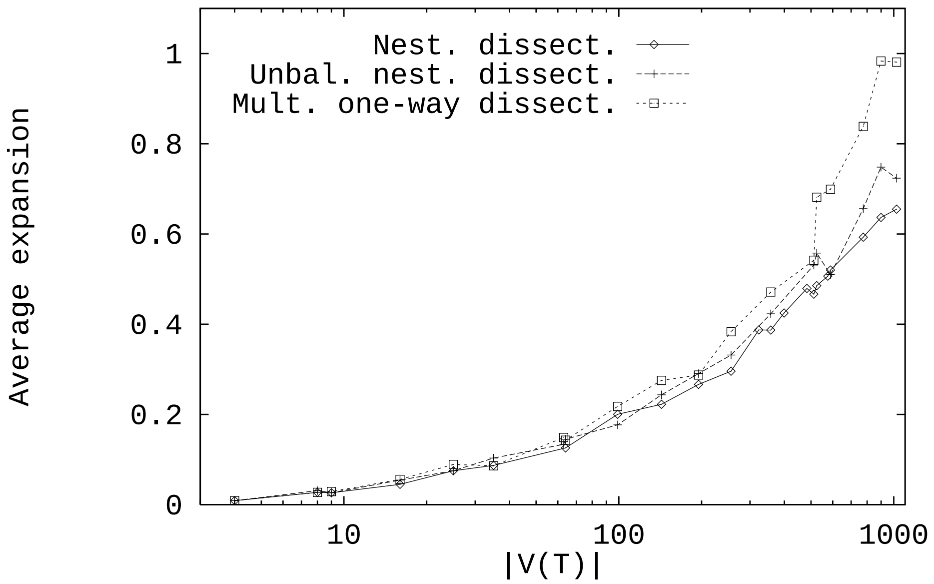


Figure 20: Average expansion of the mapping of graph *BRACKET* onto bidimensional grids decomposed algorithmically by nested dissection, unbalanced (one third/two thirds) nested dissection, and multiple one-way dissection.

these locality properties behave rather badly with the DRB algorithm. This is for instance the case for the FFT target architecture.

For the computation of mapped decompositions, cut minimization is more important than load balance. Therefore, a large load imbalance tolerance should be used, and bipartitioning methods that seek strict load balance, such as the exactifying method, should be avoided.

8 The Scotch software package

Scotch [22] is a software package for static mapping which embodies the algorithms developed within the Scotch project. Apart from the mapper itself, the Scotch package contains programs to build and test source graphs, compute target graph decompositions, and visualize mapping results. Advanced command-line interface and vertex labeling capabilities make them easy to interface with other programs. See [20] for details.
The mapper can map any weighted source graph onto any weighted target graph,

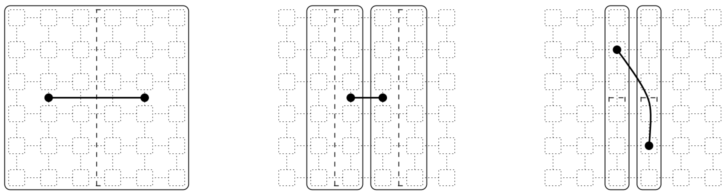


Figure 21: Decomposition of the $M_2(6,6)$ architecture by multiple one-way dissection, and its influence on the distance function. The variability of the distance function decreases as bipartitionings are performed in the first dimension, but roughly increases when the first bipartitioning is performed in the second dimension.

or even onto disconnected subgraphs of a given target graph, which is very useful in the context of multi-user parallel machines. On these machines, when users request processors in order to run their jobs, the partitions allocated by the operating system may not be regular nor connected, because of existing partitions already attributed to other people. With SCOTCH, it is possible to build a target decomposition corresponding to this partition, and therefore to map processes onto it, automatically and regardless of the partition shape.

The SCOTCH 3.0 academic distribution may be obtained from the SCOTCH `WWW` page at `http://www.labri.u-bordeaux.fr/~pelegrin/scotch/`, or by `ftp` at `ftp.u-bordeaux.fr` in directory `/pub/Local/Info/Software/Scotch`. The distribution file, named `scotch_3.0A.tar.gz`, contains the executables for several machines and operating systems, along with documentation and sample files. A collection of test graphs in our format, gathered from other packages and from individuals, is also available from the `WWW` page.

9 Comparison to other partitioning and mapping software packages

When mapping onto the complete graph, SCOTCH behaves as a standard graph partitioner. Table 4 summarizes edge cuts that we have obtained for classical test graphs, compared to the ones computed by CHACO [12] and METIS [16]. Since not accounting for the target topology generally leads to worse performance results of the mapped applications [10, 13] due to long-distance communication, static mapping is more attractive than strict partitioning for most communication-intensive applications. Recently, CHACO has recently gained static mapping capabilities by the addition of a feature called *terminal propagation* [14], which is similar to the accounting for cocycle edges that we do in f'_C for our DRB algorithm. Tables 5 and 6 summarize some results that have been

Graph	Chaco 1.0			MeTiS 2.0		
	K(64)	K(128)	K(256)	K(64)	K(128)	K(256)
4ELT	**2928**	**4514**	**6869**	2965	4600	6929
BCSSTK30	241202	318075	423627	**190115**	**271503**	384474
BCSSTK31	65764	98131	141860	**65249**	**97819**	**140818**
BCSSTK32	106449	153956	223181	**106440**	**152081**	**222789**
BRACKET	34172	46835	66944	**29983**	**42625**	60608
PWT	9166	12737	18268	**9130**	**12632**	**18108**
ROTOR	53804	75140	104038	**53228**	**75010**	**103895**

Graph	Scotch 3.0		
	K(64)	K(128)	K(256)
4ELT	2941	4604	6900
BCSSTK30	194539	277122	**382375**
BCSSTK31	70275	102250	143212
BCSSTK32	112846	160429	226991
BRACK2	30270	42743	**60583**
PWT	9286	12887	18366
ROTOR	55511	77136	105006

Table 4: Edge cut produced by Chaco 1.0, MeTiS 2.0, and Scotch 3.0 for partitions with 64, 128, and 256 blocks (Chaco and MeTiS data extracted from [15]).

obtained by Chaco 2.0 with terminal propagation and by Scotch 3.0 when mapping graphs *4ELT* and *BCSSTK32* onto hypercubes and meshes of various sizes.

In many cases, Scotch 3.0 produces partitions whose cut and communication cost are within a few percent of the ones computed by the two other programs, and can therefore be used as a state-of-the-art graph partitioner and mapper. However, for some graphs (*e.g.* *BCSSTK32*), its results are of much less quality. This lack of quality is due to the fact that Scotch 3.0 does not use a multi-level approach, as do the two others [12, 15]. By coarsening the graphs they work on, multi-level partitioners increase the capability of their local partitioning algorithms to take advantage of topological properties that were else of a too global level for them to deal with, and thus greatly improve their ability to avoid local minima of the cost function. Consequently, we are currently developing a multi-level method for Scotch, which will be available in version `3.1`. The first results that we have obtained with this new version confirm our analysis, since the cut values that we get become equivalent to the ones of the two other software packages, and outperform them in two thirds of the tested cases [21].

Target	CHACO 2.0-TP		SCOTCH 3.0	
	cut	f_C	cut	f_C
H(1)	168	168	143	**143**
H(2)	412	484	402	**403**
H(3)	796	863	693	**761**
H(4)	1220	1447	1191	**1314**
H(5)	1984	2341	2027	**2307**
H(6)	3244	**3811**	3280	3858
H(7)	5288	6065	5131	**6049**
M_2(5,5)	1779	**2109**	1929	2423
M_2(10,10)	4565	**6167**	4612	6361

Table 5: Edge cut and communication cost produced by CHACO 2.0 with Terminal Propagation and by SCOTCH 3.0 for mappings of graph *4ELT* onto several target architectures (CHACO data extracted from [11]).

Target	CHACO 2.0-TP		SCOTCH 3.0	
	cut	f_C	cut	f_C
H(1)	5562	**5562**	11381	11381
H(2)	15034	**15110**	24711	26092
H(3)	26843	**27871**	44736	48872
H(4)	49988	**53067**	63273	72802
H(5)	79061	**89359**	93197	115148
H(6)	119011	**143653**	133788	167356
H(7)	174505	**218318**	184400	240890
M_2(5,5)	64156	**76472**	77504	117794
M_2(10,10)	150846	**211672**	169165	287467

Table 6: Edge cut and communication cost produced by CHACO 2.0 with Terminal Propagation and by SCOTCH 3.0 for mappings of graph *BCSSTK32* onto several target architectures (CHACO data extracted from [11]).

10 Conclusion

In this paper, we have presented the Dual Recursive Bipartitioning (DRB) algorithm for static mapping that we have developed. Several studies have been carried out in order to validate the design choices that we made, and to evaluate the influence of the parameters of the algorithm on the quality of mappings. In particular, we have shown that a depth-first traversal of the bipartitioning tree is more suitable to map graphs that are strongly connected, and that breadth-first-like sequencing is most efficient for graphs that exhibit great locality, such as finite-element meshes. We have evidenced that indexing the gains of the Fiduccia-Mattheyses graph partitioning algorithm on a logarithmic scale does not undermine its performance, and allows it to store and retrieve huge gain values in constant time. We have also shown that the decompositions of target architectures must be such that the variations of the distance function decrease with time, in order to preserve the local nature of the DRB algorithm. Then, we have outlined the capabilities of SCOTCH, a software package for static mapping which implements the DRB algorithm and is able to map any weighted source graph onto any weighted target graph. The mappings and partitions produced by SCOTCH 3.0 are globally equivalent in quality to the one of the most efficient software packages known, although for some cases its lack of a multi-level method impedes its performance.

Work in progress includes the development of new graph bipartitioning methods, and in particular of multi-level schemes, as in [15]. The first results obtained to date with the new version show an average gain in quality of about 10 percent compared to the mappings computed by SCOTCH 3.0, which makes it more efficient than CHACO and METIS in two thirds of the tested cases. SCOTCH is currently being evaluated to decompose unstructured meshes into domains for parallel aerodynamics codes that run on the Cray T3D. We expect this study to help us determine the characteristics of efficient mappings with respect to the type of numerical method used, in order to develop suitable bipartitioning strategies. A nested-dissection ordering code for a parallel direct block solver is also being developed, based on the graph partitioning library that makes up the core of our mapping program.

References

[1] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[2] A. Bel Hala. Congestion optimale du plongement de l'hypercube $H(n)$ dans la chaîne $P(2^n)$. *RAIRO Informatique Théorique et Applications*, 27(4):1–17, 1993.

[3] P. Charrier and J. Roman. Partitioning and mapping for parallel nested dissection on distributed memory architectures. In *LNCS 634 – Proceedings of CONPAR'92*, pages 295–306. Springer-Verlag, September 1992.

[4] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[5] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractablility: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[7] J. A. George and J. W. H. Liu. *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

[8] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[9] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. Technical Report TR 87-803, Cornell University, January 1987.

[10] S. W. Hammond. *Mapping unstructured grid computations to massively parallel computers*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New-York, February 1992.

[11] B. Hendrickson. Personal communication. July 1996.

[12] B. Hendrickson and R. Leland. The CHACO user's guide – version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[13] B. Hendrickson and R. Leland. An empirical study of static load balancing algorithms. In *Proceedings of SHPCC'94, Knoxville*, pages 682–685. IEEE, May 1994.

[14] B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing Data Locality by Using Terminal Propagation. In *Proceedings of the 29^{th} Hawaii International Conference on System Sciences*. IEEE, January 1996.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. TR 95-035, University of Minnesota, June 1995.

[16] G. Karypis and V. Kumar. METIS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System – Version 2.0.* University of Minnesota, June 1995.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitionning graphs. *BELL System Technical Journal*, pages 291–307, February 1970.

[18] D. M. Nicol. Rectilinear partitioning of irregular data parallel computations. *Journal of Parallel and Distributed Computing*, 23:119–134, 1994.

[19] F. Pellegrini. Static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of SHPCC'94, Knoxville*, pages 486–493. IEEE, May 1994.

[20] F. Pellegrini. SCOTCH 3.0 User's guide. Technical Report 1095-95, LaBRI, Université Bordeaux I, October 1995. Available at URL `http://www.labri.u-bordeaux.fr/~pelegrin/papers/scotch_user3.0.ps.gz`.

[21] F. Pellegrini. Graph partitioning based methods and tools for scientific computing. In *Proceedings of ETPSC III, Faverges de la Tour*. SIAM, August 1996. To be published.

[22] F. Pellegrini and J. Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of HPCN'96, Brussels,* LNCS 1067, pages 493–498, April 1996.

[23] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, July 1990.

[24] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.