Process Mapping onto Complex Architectures and Partitions Thereof

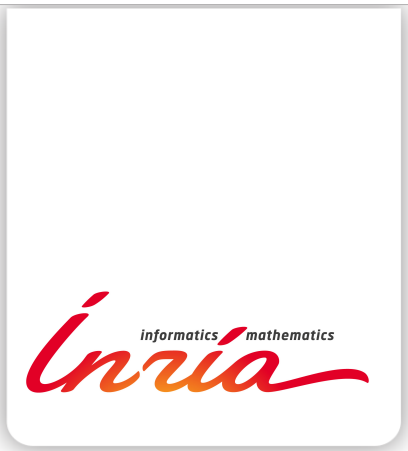
François Pellegrini, Cédric Lachat

Process Mapping onto Complex Architectures and Partitions Thereof

François PELLEGRINI, Cédric LACHAT

RESEARCH REPORT
N° 9135
December 2017
Project-Team TADaaM

ISSN 0249-6399 ISRN INRIA/RR--9135--FR+ENG



Process Mapping onto Complex Architectures and Partitions Thereof

François Pellegrini*, Cédric Lachat†

Project-Team TADaaM

Research Report n° 9135 — December 2017 — 16 pages

Abstract: Data locality is a critical issue in order to achieve performance on today's high-end parallel machines. As these machines are highly non-uniform, distributing computations across their processing elements does not only require to minimize inter-process communication, but also to favor local communication over distant communication. For that purpose, static and/or dynamic (re)mapping tools have been devised, that allow one to map process graphs onto architecture graphs describing the topology and architectural features of such machines. However, in practice, the real problem to solve is to map a process graph onto possibly disconnected parts of a non-uniform parallel machine, such as a set of nodes provided by some batch scheduler.
This paper presents a set of algorithms to perform this task in an efficient way. Efficiency is achieved thanks to a multilevel description of target architectures. All the presented algorithms have been implemented in the Scotch static mapping software. Experiments evidence the quality of the produced mappings.

Key-words: Graph partitioning, static mapping, process mapping.

* francois.pellegrini@labri.fr
† cedric.lachat@inria.fr

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Placement de processus sur des architectures complexes et des partitions d'icelles

Résumé : La localité des données est une question critique afin d'obtenir des performances sur les machines massivement parallèles actuelles. Comme ces machines sont hautement non-uniformes, distribuer efficacement les calculs sur leurs éléments de traitement ne nécessite pas seulement de minimiser la communication inter-processus, mais aussi de favoriser la communication locale par rapport à la communication distante. Dans ce but, des outils de (re)placement statique et/ou dynamique ont été conçus, qui permettent de placer des graphes de processus sur des graphes d'architecture représentant la topologie et les caractéristiques architecturales de ces machines. Cependant, en pratique, le vrai problème à résoudre est de placer un graphe de processus sur des parties potentiellement déconnectées d'une machine parallèle non uniforme, telles que des ensembles de nœuds attribués par un ordonnanceur *batch*.

Cet article présente un ensemble d'algorithmes effectuant cette tâche d'une façon efficace. L'efficacité est obtenue grâce à une description multi-niveaux des architectures cibles. Tous les algorithmes présentés ici ont été implémentés dans le logiciel de placement statique Scotch. Des expérimentations illustrent la qualité des placements produits.

Mots-clés : Partitionnement de graphe, placement statique, placement de processus.

1 Introduction

Data locality is a critical issue in order to fully exploit petascale and exascale parallel architectures. Such architectures are highly non-uniform: transferring data to a processing element located on the same chip is orders of magnitude less expensive than sending data to a processing element located in a remote cabinet. Consequently, data allocation must be performed in a way that does not minimize inter-process communication uniformly across all processing elements, but favors local communication over distant communication, even at the price of an increase in overall communication volume.

In many application domains, such as scientific computing, data allocation can be performed using graph partitioning [**?**, **?**]. Data are modeled as the vertices V_S of a graph $G_S(V_S, E_S)$, often called *process graph*, the edges E_S of which represent computational dependencies. Graph vertices are partitioned into as many parts as the prescribed number of processing elements, possibly with respect to their respective compute power, while reducing the number of edges connecting vertices that belong to different parts. Graph mapping is a generalization of the graph partitioning problem, in which the vertices of the process graph are placed on the vertices V_T of a *target graph* $G_T(V_T, E_T)$. In this case, the metric to be minimized is not the number of cut edges, but a cost function that may take into account the *dilation* of the edges, that is, the length of the path, in the target graph, that connects two mapped vertices that are neighbors in the process graph.

The mapping problem received some attention during the 1970's, with the advent of distributed-memory multi-computers exhibiting highly non-uniform access costs. Then, interest for it faded in the 1990's, as computer architects succeeded in designing hardware mechanisms that alleviated this non-uniform behavior. With the return into force of highly non-uniform architectures designed for petascale and exascale computing, this problem is now considered again as a key issue [**?**, **?**].

Mapping and partitioning are NP-hard problems [**?**], except in very special cases [**?**]. The first heuristics designed to address the partitioning and mapping problems were cubic or quadratic in $|V_s| + |E_S|$ [**?**, **?**]. Then, because of the increasing size of process graphs, new algorithms were proposed, that were quasi-linear in $|V_s| + |E_S|$ [**?**]. However, the complexity of these algorithms also depends in $|V_T|$. While the run time complexity of these algorithms with respect to this parameter is often low, e.g. in $\log(|V_T|)$ for a recursive mapping algorithm [**?**], they manage data structures that are quadratic in $|V_T|$. This was also the case for Scotch, the graph mapping software that we have been developing for more than two decades.

This report presents algorithmic improvements that dramatically reduce the complexity of the data structures designed to represent target architectures in Scotch. They allow it to address machine sizes of petascale and exascale classes. They extend important features, such as the ability to map onto irregular and/or disconnected parts of these architectures, which is necessary to manage sets of nodes returned by batch schedulers.

This report is organized as follows. In Section 2, we perform a quick state of the art of the mapping problem, focus on the mapping methods used in Scotch, and highlight their current drawbacks. In Section 3, we present the core of our contribution: the multilevel description of generic target architectures, and the way they are implemented. Then, in Section 4, we evidence the benefits of these new data structures on illustrative examples. Then comes the conclusion.

2 State of the art

It is not the purpose of this section to perform an extensive survey of the mapping problem, which already exists in the literature [**?**, **?**]. Rather, we will present a brief overview of mapping methods

based on a recursive and/or hierarchical framework, as they possess the strongest connections with our own work.

Faced with the challenge of computing a k-way partition or mapping, many authors followed the "divide and conquer" approach, through recursive bipartitioning [**?**, **?**, **?**, **?**]. Indeed, doing so simplifies heuristics and associated data structures, because in the bipartitioning case vertex moves across parts do not require to decide to which other parts to move them, and degenerate into straightforward vertex swaps.

In the mapping case, this amounts to a scheme in which a target architecture, such as a hypercube [**?**], is partitioned into two pieces, after which the graph to map is subsequently split into two parts, which are assigned to the two architecture pieces. This procedure is recursively applied until architecture parts are restricted to a single processing element. In order to improve locality, at each step, the graph bipartitioning algorithm does not try to reduce the cut between the two parts, but to minimize a cost function that accounts for the dilation of the edges that are mapped onto other parts. In the case of a hypercube, the dilation is computed as the Hamming distance between the two considered sub-hypercubes, that is, a shortest-path distance [**?**, **?**].

Overall, the aim of these recursive algorithms is to optimize a cost function that is the sum of the weighted dilation of all edges:

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

where $w_S(e_S)$ is the weight of some edge e_S of S, $\tau_{S,T} : V_S \longrightarrow V_T$ is the application that maps vertices of S onto vertices of T, and $\rho_{S,T} : E_S \longrightarrow \{E_T\}^*$ associates a path in T, of length $|\rho_{S,T}(e_S)|$, to any edge e_S of S.

The strong positive correlation between values of this function and effective execution times has been experimentally verified on many generations of distributed-memory computer systems [**?**, **?**, **?**].

Our first work on the subject consisted in generalizing the aforementioned work of Ercal *et al.* [**?**] by abstracting the notions of "target architecture", to extend it to arbitrary topologies), of "part of a target architecture" obtained by recursive bipartitioning, and of "distance" between any two such parts. This led to the definition of the *Dual Recursive Bipartitioning* algorithm, which we are going to describe below.

2.1 The Dual Recursive Bipartitioning algorithm

The mapping algorithm that we originally designed for Scotch is called *Dual Recursive Bipartitioning* (DRB) [**?**].

2.1.1 Principle of DRB

The purpose of the DRB algorithm is to map neighbor vertices of the source graph as closely as possible to each other in the target architecture. However, it does not compute routings, that is, the paths in the target graph that correspond to every source edge. We consider that modeling the behavior of the communication system of every possible target architecture was a task way too difficult, all the more that the relevant information may not be made available by the vendor or may change without notice due to vendor software releases. Moreover, for time-shared and/or batch-scheduled systems, the behavior of the communication subsystem may evolve dynamically, according to resource requirements of concurrent jobs. Consequently, we decided to leave the effective handling of communication to the communication subsystem of the target machine. Nevertheless, we assume that the operator of the system is able to model

the features and average performance of the interconnection network on the form of a weighted graph, that can be used by application software users willing to run their software on the system.

The DRB algorithm is similar in nature to that of [**?**]. It is based on a *divide & conquer* approach, and proceeds by recursive allocation of subsets of processes to subsets of processors, till the processor subsets are restricted to one element or the process subsets are empty. At each step, it bipartitions a subset of processors, also called a *domain*, into two disjoint subdomains, and calls a *graph bipartitioning* algorithm to assign the vertices of its associated *process subgraph* onto the two subdomains with respect to our cost function, as sketched below. When the domain is a single-vertex domain, called *terminal domain*, recursion stops and the processes of the associated process subgraph are mapped onto this target vertex.

Algorithm 1 The Dual Recursive Bipartitioning algorithm.

```
mapping (P, D)
Process_graph   P;
Target_domain   D;
{
  Process_graph   P0, P1;
  Target_domain   D0, D1;

  if (|P| == 0)          /* If no processes assigned to this domain. */
    return;              /* Nothing to do. Processors will be idle.  */

  if (|D| == 1) {        /* If one processor in D */
    result (P, D);       /* P is mapped onto it.  */
    return;
  }

  (D0, D1) = domain_bipartition (D);
  (P0, P1) = graph_bipartition  (P, w(D0), w(D1)); /* Weighted bipartition. */

  mapping (D0, P0);      /* Perform recursion. */
  mapping (D1, P1);
}
```

In subsequent versions of Scotch, the set of available mapping algorithms has been extended. In particular, a multilevel framework has been implemented, as initially proposed by [**?**], to improve speed and mapping quality. However, even in this framework, the initial k-way mappings of the coarsest graph is still computed using the DRB algorithm.

2.1.2 Abstractions of target architectures

The DRB algorithm relies on the ability to define five main objects:

- a *domain* structure, which represents some part of the target architecture, onto which some part of a process graph (that is, a subgraph) is to be mapped;
- a *domain bipartitioning function*, which, given a domain, bipartitions it into two disjoint subdomains separated by a small cut, and consequently high communication locality within each of the subdomains. Subdomains do not need to be of the same size;
- a *domain distance function*, which gives, in the target architecture, 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. This domain distance function is

used to evaluate the communication cost function to minimize, by allowing graph bipartitioning algorithms to estimate the dilation of the edges linking graph vertices which belong to different domains;

- a *weighted graph* structure, which represents a part of the process graph to be mapped;
- a *weighted graph bipartitioning function* which, given the respective weights of two subdomains, and a process (sub)graph, bipartitions the latter into two disjoint process sets, the relative weights of which match those of the two subdomains.

All these routines are seen as black boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning function.

As a matter of fact, according to the topology of the target architecture, the results of the domain handling routines can be algorithmically computed at run-time for regular architectures, or be extracted from pre-computed tables in the case of irregular architectures (see Section 2.3).

2.2 Execution scheme

The DRB algorithmic framework is greedy by nature, since the mapping of a process onto 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 non-leaf node of which corresponds to a bipartitioning job, that is, the bipartitioning of both a target architecture subdomain and its associated process subgraph.

In the case of a depth-first sequencing, as written in the above sketch, bipartitioning jobs called in the left branches have no information on the distance to vertices to be processed by the right branches. On the contrary, sequencing the jobs according to a by-level (breadth-first) travel of the tree allows that, at any level, any bipartitioning job may have information on the subdomains to which all the processes have been allocated during the previous level. In the latter case, when deciding in which subdomain to put a given process, a bipartitioning job can account for the communication costs induced by the neighbor processes, whether they are handled by the job itself or not, since it can estimate the dilation of the corresponding edges. This results in an interesting feed-back effect: once an edge has been caught in a cut between two subdomains, the distance between its end vertices will be accounted for in the communication cost function to be minimized, and subsequent bipartitioning jobs will thus be more likely to keep these vertices close to each other, as illustrated in Figure 1. The partial cost function to be optimized during each bipartitioning job is:

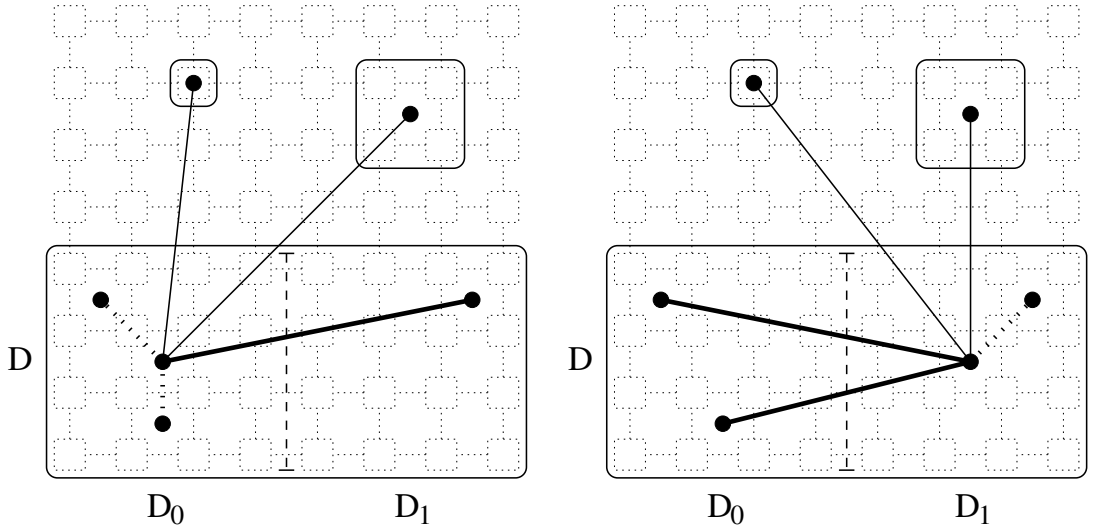
$$f'_C(\tau_{S,T}, \rho_{S,T}) \stackrel{\text{def}}{=} \sum_{\substack{v \in V(S') \\ \{v,v'\} \in E(S)}} w_S(\{v,v'\}) \, |\rho_{S,T}(\{v,v'\})| \ ,$$

which accounts for the dilation of edges internal to subgraph S' of S as well as for the one of edges which belong to the cocycle of S' (that is, edges that have exactly one of their ends in S').

An additional advantage of using breadth-first traversal is that subdomains that are assigned to end vertices maintain equivalent sizes during all the recursive bipartitioning process. This respects the distance homogeneity requirement and gives the algorithm more coherence [**?**].

To allow for some flexibility in sequencing the bipartitioning jobs, all pending jobs are stored into a queue. The execution of a bipartitioning job results in the creation of at most two new jobs, which are en-queued in turn. The next job to be processed is selected according to a user-configurable policy. For instance, depth-first traversal can be implemented by always selecting the job with the highest level, and breadth-first traversal by always selecting the queue head. The

default method implemented in Scotch consists in selecting the job associated with the biggest subgraph. The rationale for this breadth-first-like method is to provide more accurate partial mapping information to neighboring jobs, because larger jobs are likely to possess more cocycle edges, that is, edges that connect them to the rest of the process graph. Its behavior differs from regular breadth-first sequencing in the case of weighted graphs, where some subgraphs of the same level may contain much less vertices (yet heavier ones) than others. Considering graph size is a shortcut that avoids computing the exact number of cocycle edges for each subgraph (or, better, the sum of their weights, that would have a higher impact on the cost function), assuming a homogeneous degree distribution.



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.

2.3 Shortcomings of current DRB implementation

Before the works we present here, the implementation of the DRB algorithm in Scotch had two shortcomings: one regarding decomposition-defined architectures, and the other regarding algorithmically-defined architectures. Both are presented below.

2.3.1 Decomposition-defined architectures

Decomposition-defined architectures are the most generic way to represent (non-regular) target architectures is Scotch. In order to use this architecture, the user just has to provide a weighted graph, the vertex weights of which represent processing power, and edge weights of which represent link latency/cost (that is, the bigger an edge weight is, the more expensive it is for data to traverse it).

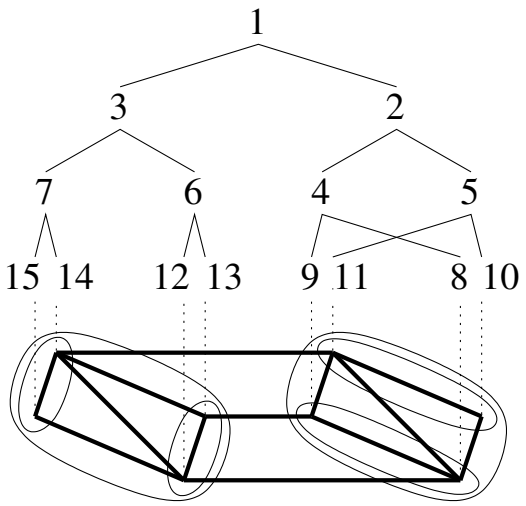
This graph is processed in two consecutive phases. In the first phase, a recursive bipartitioning process is applied to the graph, until all parts are of size 1. During each bipartitioning step, a

post-processing method is called, so that the sizes of the two parts are as even as possible. The purpose of this stage is to avoid empty parts, that would prevent the obtainment of a complete bipartitioning tree. This tree defines all the domains that will be handled by the DRB mapping process. They are labeled in the following way: the domain representing the whole graph is labeled as domain 1; then, the two parts of some domain i are labeled with labels $2i$ and $2i+1$. For instance, Figure 2.a represents the labeled tree obtained when recursively bipartitioning the 8-vertex De Bruijn graph UB$(2,3)$. Note that the recursive bipartitioning tree of a target graph can be reconstructed easily from the terminal domain labels associated with each of its vertices.

In the second phase, an inter-domain distance matrix is computed, in a bottom-up way. Firstly, the distance between any pair of vertices is computed by means of successive breadth-first search algorithms (we do not use Dijkstra's algorithm for that purpose, see rationale in Section 3.1), and stored in the distance matrix. Then, traversing the target graph bipartitioning tree in a bottom-up way, the distance between any two non-terminal subdomains is computed by averaging the distance between all pairs of their son subdomains (hence, at most four values are merged at each step).

The resulting target architecture data structure provides all the architecture abstractions listed in Section 2.1.2: a domain representation and labeling, a domain bipartitioning function (which returns the two sons $2i$ and $2i+1$ of some non-terminal domain i), and a domain distance function.

All these elements are stored in the Scotch intermediary file format for decomposition-defined target architectures, called "`deco 0`". The contents of this file for the De Bruijn graph of Figure 2.a are shown in Figure 2.b. It is an architecture with 8 vertices (that is, terminal domains), and 15 domains. The first block of lines records on each line the vertex index, vertex weight and terminal domain label of every target graph vertex. The second block is the lower part of the (symmetric) distance matrix between all graph vertices. The averaged part of the matrix for non-terminal domains is not stored, as it would quadruple matrix data, but is recomputed after loading.



a. Recursive coarsening tree of UB$(2,3)$.

```
deco 0
8 15
0 1 15
1 1 14
2 1 13
3 1 11
4 1 12
5 1 9
6 1 8
7 1 10
```

```
1
2 1
2 1 2
1 1 1 2
3 2 1 1 2
2 2 2 1 1 1
3 2 3 1 2 2 1
```

b. Derived "`deco 0`" file.

Figure 2: Target decomposition file for UB$(2,3)$. Terminal domain numbers associated with every target vertex define a unique recursive bipartitioning of the target graph.

Due to the pre-computation and exhaustiveness of the distance matrix, the compute time and storage of which are in $O(|V_T|^2)$, distance values between any two subdomains of the bipartitioning tree can be provided in $O(1)$ during the mapping process.

Decomposition-defined target architectures can be restricted to a subset of the graph vertices, by providing a vertex list that contains the indices of the graph vertices that are available as mapping targets. It differs from providing a graph restricted to the said vertices, because in the latter case disconnected components would be considered at infinite (unknown) distance from each other, since there would exist no path between them. When a vertex list is provided, the bipartitioning tree is computed such that its nodes only bear the weights of vertices that are possible mapping targets, while distance matrix values are unchanged between the selected vertices, because paths between them are still the same in the graph.

The vertex list feature has several uses. Firstly, it allows one to describe an architecture made of routers and processing elements, and to restrict mappings to processing elements only. Secondly, it allows a program to compute data distributions that take into account the location of the nodes that have been assigned to it by a batch scheduler. Yet, in such cases, network contention induced by parallel programs that may run on other, separating parts of the machine is not taken into account by default. To account for it, the weights of the links of the target graph have to be tuned according to measured contention. However, such finely tuned data distributions have to be recomputed when interweaved, perturbating jobs change of communication behavior and/or terminate. Such dynamic aspects are out of the scope of this report. We leave to the user to take them into account in the target graph description if they want to.

While the decomposition-defined architecture data structure is very flexible, it is not scalable, since its storage is quadratic in the number of vertices, due to the explicit distance matrix. Hence, it cannot be used to represent parallel machines of sizes bigger than a few thousands of processing elements, while high-end machines comprise more than several hundred thousands of them.

2.3.2 Algorithmically-defined architectures

Algorithmically-defined architectures are another possibility to represent target architectures in Scotch. In this case, the code pieces that define the architecture and subdomain data structures, and the routines that perform subdomain bipartitioning and compute inter-domain distance, are included in Scotch at compile time. Such architectures comprise hypercubes, d-dimensional grids and tori, hierarchical architectures (called "tree-leaf", because the processing elements are the leaves of a tree structure whose non-leaf nodes are assumed to be routers), etc.

For instance, hypercube architectures are extremely easy to represent. A hypercube of dimension d is described by this very dimension value. A subdomain of a hypercube of dimension d is a sub-hypercube of dimension d', with $d \geq d' \geq 0$. It is described by two values: the dimension of the subdomain, and the Hamming coding of the bits that have already been set to label the subdomain. This representation provides a natural way to bipartition a domain. For instance, the domain that represents the whole of a hypercube of dimension d is $(d, --)$. The two sons of domain $(d, --)$ are $(d-1, 0_b)$ and $(d-1, 1_b)$, the two sons of the latter are $(d-2, 10_b)$ and $(d-2, 11_b)$, and so on. When the first number is equal to zero, the subdomain is restricted to a single vertex, the label of which is the d-bit, second number. The distance between any two domains is the Hamming distance between the fixed bits in the two domain labels, plus half of the number of unknown bits (to average the unknown Hamming distance).

Grid-like architectures comprise multidimensional meshes and tori. The only difference between these two architectures concerns inter-domain distance computation. Grid-like architectures of dimension d are defined by a vector of size d holding the number of processing elements in each dimension. A domain is defined by its inclusive minimum and maximum coordinates in

the d-dimensional grid. A domain is bipartitioned by cutting it along its largest dimension. The distance between two domains is the Manhattan distance between the centers of the domains. In the case of tori, this distance is usually shorter than in the case of grids, because of the existence of wrap-around edges in every dimension.

One can note that, from a software design point of view, the decomposition-defined architecture handling module is an instance of algorithmically-defined architecture. The architecture data structure contains the subdomain weight vector and inter-subdomain distance matrix described in Section 2.3.1, and the domain bipartitioning and domain distance computation routines extract relevant information from these data structures.

Unlike decomposition-defined architectures, algorithmically-defined architectures can be scalable if their implementation is adequate. However, before the works presented in this paper, they had a main drawback: it was not possible to provide a vertex list to map onto a (possibly disconnected) part of an architecture of this kind.

This rendered the use of algorithmically-defined architectures less interesting for applications running on shared-use, large-scale systems. Indeed, while a subset of a d-dimensional grid may be a d-dimensional grid itself, if partitions are properly organized by the system operator and/or the batch scheduler, a subset of a d-dimensional torus can never be a d-dimensional torus, because of the absence of wrap-around edges. Moreover, most batch schedulers provide irregularly-shaped, and even disconnected, sets of nodes.

3 Multilevel target architectures

As seen above, target architectures in Scotch had two drawbacks, which prevented their use on high-end supercomputers: decomposition-defined architectures were not scalable, while algorithmically-defined architectures did not allow one to map onto an irregular subset of such an architecture. The aim of this work is to solve these two issues concurrently.

3.1 Rationale for a solution

In the case of decomposition-defined architectures, inter-domain distance computation is the critical issue. Computing exact distances between vertices of a million-vertex graph without storing a distance matrix would require running Dijkstra's algorithm each time such a distance is needed. The issue is also to compute distances between subdomains that are not restricted to single vertices, but derive from some recursive bipartitioning of the initial target graph. Consequently, a hierarchical, tree-like representation is still mandatory.

Top-down decomposition such as recursive bipartitioning is a way to manage target architectures hierarchically. However, distance computation is not performed using a top-down approach. As seen in Section 2.3.1, in the current implementation of the decomposition-defined architecture, once the recursive decomposition has been performed completely, the distance matrix is computed in a bottom-up way.

Instead of computing the bipartitioning tree first, and then traversing the tree in a bottom-up way, one can consider a more efficient, direct bottom-up approach. It consists in creating pairs of vertices by mating neighboring vertices, and collapsing every vertex pair, until the entire graph is reduced to a single, root vertex. This recursive coarsening approach is in fact the very one followed by multilevel methods.

3.2 Coarsening-based multilevel representation

In order to illustrate the construction of the bipartitioning hierarchy, let us consider the 4×2 bi-dimensional grid target architecture drawn as the top left graph in Figure 3. Out of the 8 vertices of this graph, 5 only are available as mapping destinations: those that are labeled with a 1. Other vertices are labeled with a 0.

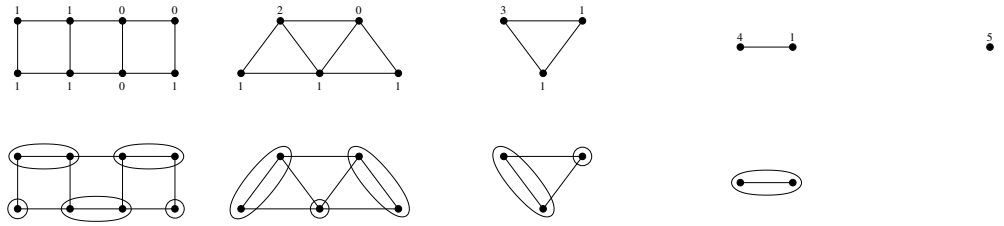


Figure 3: Locality-based clustering of a labeled 4×2 grid target architecture (upper left). A matching of the graph is computed (bottom left), then the graph is coarsened (second upper left), then a second matching is computed (second bottom left), and so on.

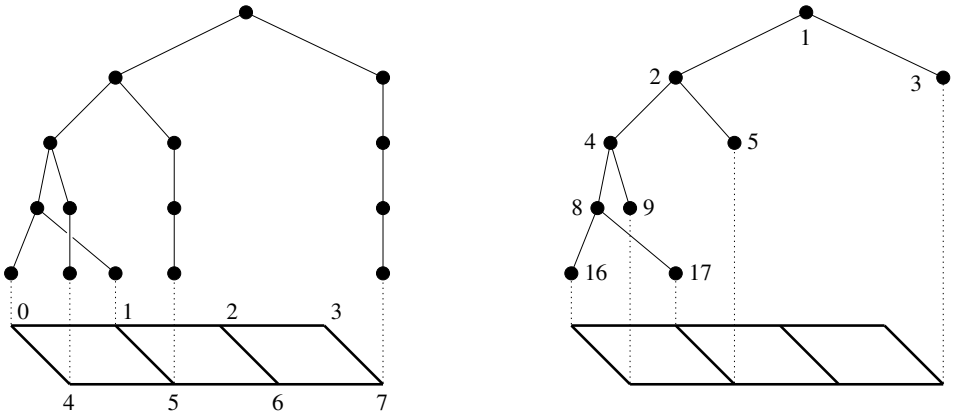
Using a heavy-edge matching algorithm on this graph yields the matching displayed at the bottom left of Figure 3. Collapsing the mated edges yields in turn the second graph from the left in the upper part of the figure. The labels of the collapsed vertices are computed by adding the labels of the finer vertices. They represent the number of vertices available for mapping in the produced clusters. The matching and coarsening process is repeated until the graph is reduced to a single vertex.

The above recursive coarsening process can be described in the form of a tree, from which nodes with zero weight are discarded. In most cases, the recursive coarsening tree is not equivalent to a recursive bipartitioning tree, because when a vertex is coarsened to itself, it produces a node that has only one son. In order to obtain a regular recursive bipartitioning tree, the recursive coarsening tree must be compacted, as illustrated in Figure 4.

The resulting sequence of recursive bipartitionings is illustrated in Figure 5. One can see that, because the coarsening process favors locality, and vertex 7 is isolated, the first bipartitioning is highly imbalanced. This is consistent with the way to handle such partitions: a small set of lightly connected processes has to be separated from the process graph, while the remaining, tightly connected subgraph will be placed on a more tightly connected set of processors.

3.3 Application to algorithmically-defined architectures

The multilevel target architecture framework that we have presented above requires to compute successive matchings of the architecture graph. While this is a natural operation for decomposition-defined architectures, for which the graph is available by definition, algorithmically-defined architectures are not based on a graph representation; the very purpose of these architectures was indeed to avoid the cost of managing one. Also, using a standard, pseudo-random based matching on these architecture would lead to irregularly-shaped domains, which could not be represented by regular domain data structures. Consequently, the matching of algorithmically-defined architectures has been implemented by means of architecture-specific, deterministic, routines.



a. Recursive coarsening tree obtained from Figure 3. **b.** Derived recursive bipartitioning tree.

Figure 4: Compacting of a recursive coarsening tree (a) into a recursive bipartitioning tree (b), by merging only sons to their fathers. Once the tree has been compacted, domain labels can be associated with every node, in a top-down way.

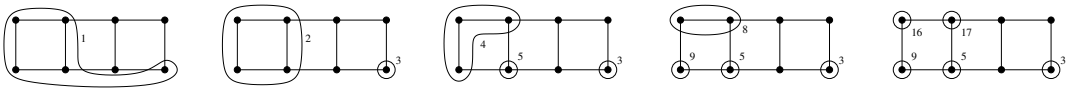


Figure 5: Sequence of recursive bipartitions of the target architecture of Figure 3 that will be performed during the DRB algorithm. Domains are labeled with their respective numbers, up to their terminal domain numbers.

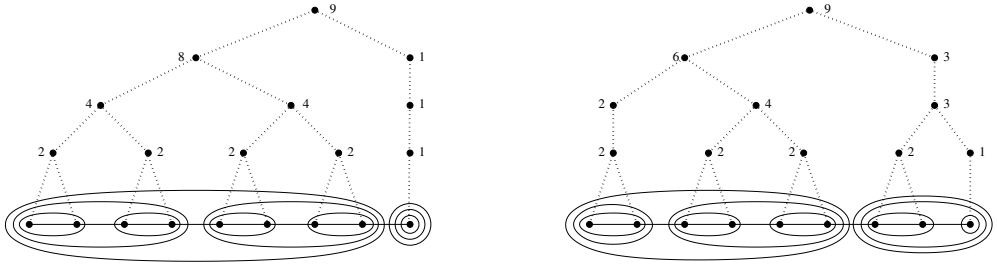
For instance, the hypercube architecture is easily coarsened by performing a projection, dimension after dimension: at stage i, all vertices the binary representation of which differ only by their i^{th} bit are merged together.

The coarsening of grid-like architectures is performed dimension after dimension, on a round-robin basis: all dimensions are coarsened one after the other. In the case when a dimension is odd, the algorithm records the starting point of the previous matching in this dimension, so as to alternate it every other time, in order to reduce imbalance in the coarsening tree, as depicted in Figure 6.

Managing matchings by means of tailored routines allows one to optimize the construction of the recursive coarsening tree. Indeed, there is no need to perform the coarsening of parts of the target architecture that will not participate in determining the locality of target vertices that will be mapping destinations. For instance, for grid-like architectures, the coarsening process can be restricted to a bounding box, the extreme coordinates of which are computed from the coordinates of all the vertices of the vertex list.

3.4 Application to decomposition-defined architectures

In the case of decomposition-defined architectures, matchings can be computed easily, because the original target graph is provided by the user. All of the necessary algorithms already exist in



a. Matching starting from the leftmost vertex only.

b. Matching starting alternatively from the leftmost and rightmost vertices.

Figure 6: Successive regular matchings along a grid dimension of odd sizes. Starting the matching always from the same side yields unbalanced bipartitioning trees (a), while alternating the start side reduces this phenomenon (b).

SCOTCH, as part of the multilevel framework for 2-way and k-way partitioning. The main issue is that of distance computation. As said above, we cannot use a pre-computed distance matrix, for scalability reasons. Similarly, we cannot use Dijkstra's algorithm on the original target graph, as it would be much too expensive. Moreover, the distance between non-terminal domains, that is, sets of vertices, has also to be computed. This might require one to define and record some "center vertex" for each of these sets in the original target graph. A trade-off must be found between storage and computation.

Since distance has only to be approximate, a practical solution is to take advantage of the family of coarser graphs that has been created in the recursive coarsening phase. The key idea is to compute or approximate the distance by traversing the graph only with the proper level of detail. We will illustrate this algorithm thanks to Figure 7.

Let us assume for instance that we want to compute the distance between two subdomains of some sizes (see level L_0 of Figure 7). A representation of this distance as a path only exists at the level at which both domains are represented by a single vertex (that is, level L_j of Figure 7). At this level, the distance between the two domains can be estimated by computing the (weighted) shortest path between these two vertices.

Yet, doing so would still raise a performance issue. In the case when the given domains are very small, the selected graph will be of a low level, and can even be the original graph itself when both domains are terminal domains. Thus, the distance computation algorithm would have to be run on this very large graph.

The solution to this problem lies in the fact that our distance function does not have to yield an exact value for long distances. The distance value has to be accurate when considering small variations of the cost function between neighboring domains, that is, when computing gains for vertices moving across neighboring domains, but can be approximated for source edges that are stretched across remote domains, because we already know that the cost will be higher than that of local distances.

Hence, we implemented the following approximate distance computation algorithm: once we have computed the level of the initial distance computation (see level L_j of Figure 7), we perform a breadth-first search from one of the two vertices in this graph. If a path towards the other vertex is not found in a given number of steps, we abort the breadth-first algorithm, climb up one level, and start a new breadth-first algorithm between their father vertices, and so on, up to the moment when we succeed in connecting them in this limited number of steps (see level L_k of Figure 7). The rationale for this algorithm is that there will always exist a level in which a

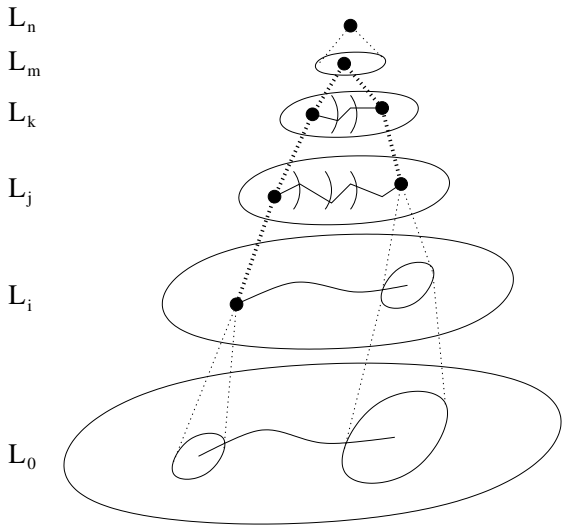


Figure 7: Multilevel computation of the distance between two subdomains of a decomposition-defined target architecture. L_0 is the finest level, that of the original target graph. Two subdomains are represented at this level. L_i is the level at which one of the subdomains is represented as a single vertex. L_j is the level at which both subdomains are represented as vertices, that is, when a single shortest path can be computed. L_m is the level at which both subdomains are merged into a common ancestor vertex. L_n is the final level, at which the whole graph is restricted to a single vertex. Circle arcs represent breadth-first traversal levels when determining the shortest path between the two vertices.

path will be found, because there always exists a common ancestor (see level L_m of Figure 7). The higher we have to go, the more the distance is approximated, but this is consistent with the fact that in this case the two vertices are far apart one from another.

In order for distances in the coarsened graph to represent a fair estimate of the distances in the original graph, graph vertices and edges are weighted in a way to represent traversal cost. Note that this distance weighting is completely independent from the weighting that is used in the context of the recursive coarsening process to perform the so-called *heavy-edge matching* mating selection algorithm [**?**]: vertex weights provided by the user are used in the heavy edge matching process to represent processing power, but are not considered in subsequent distance computations.

For distance computations, graphs are weighted in the following way. Vertex weights represent the cost of traversal of the vertices. Hence, the weight of a coarse vertex is set as the weight of the edge that connected the two merged fine vertices. Edge weights represent the cost of traversal of the edges. Hence, the weight of a coarsened edge is the average of the weights of the edges that are merged into it. The distance between two vertices is computed from the shortest weighted path as the sum of the weights of the edges and vertices that have to be traversed, plus half the weights of the end vertices. Vertices of the original, finest target graph have zero distance weight, because they have no traversal cost, and edge weights are that provided by the user as link traversal costs.

The coarser the graph, the less accurate the approximation of traversal costs becomes. Indeed, considering the traversal cost of a coarse vertex to be the cost of the edge that connects the two matched vertices, results in overestimating, on average, the distance between any two vertices.

Consider for instance an unweighted 2D grid graph, coarsened along the horizontal dimension. Every vertex of the coarsened graph has a traversal cost equal to 1, because it is the cost of traversal of the collapsed horizontal edges. Horizontal paths are accurate, because the horizontal distance between two coarse vertices located on the same row reflects the effective distance of the corresponding fine vertices in the fine graph. On the opposite, the vertical distance between two coarse vertices located on the same column is overestimated by a factor two, because each traversed vertex adds the cost of traversal of a horizontal edge, that would not be traversed on the fine graph. When the coarse grid is further coarsened along the other dimension, the anisotropy of the distance function disappears, but the overestimation remains, save for diagonal paths, for which the estimation is still accurate.

4 Experiments

We have run several sets of experiments, to show how the use of the new multilevel target architecture model allows SCOTCH to overcome the drawbacks of its past implementations.

4.1 Experiments on algorithmically-defined architectures

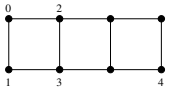
In order to handle disconnected parts of an algorithmically-defined architecture, we have created the `sub` meta-architecture. This target architecture allows one to provide a vertex list and a target architecture, from which is created an architecture restricted to the vertices of the list. The order in which the vertices are provided in the list determines the new labels of the kept vertices. This is consistent with the use of a batch scheduler that will provide processing elements in an order that will determine MPI process ranks.

For instance, Figure 8.a shows a 5-vertex sub-architecture of a 2×4 grid. The order in which the 5 vertex indices are listed makes vertex 0 of the original architecture remain vertex 0 of the restricted architecture, vertex 4 become vertex 1, and so on, as illustrated in the figure. By running SCOTCH to map the `bump` graph of Figure 8.b onto this architecture, one obtains the mapping shown in Figure 8.d, which has an edge cut of 561 and an edge dilation of 713. As one can see, SCOTCH adapted the partitions so that parts 1 and 2 do not touch, as well as parts 2 and 4, to reduce the number of long-distance edges. When performing a regular mapping of the `bump` graph into 5 parts, one obtains the more regular partition of Figure 8.c, which has an edge cut of 504. Yet, using this partition as a mapping on the architecture of Figure 8.b yields a dilation of 804. This example evidences the interest of mapping with respect to partitioning in order to improve communication locality.

The new `deco2` architecture allows SCOTCH to handle generic graphs in the same way, albeit with a much higher cost for computing distances between subdomains, due to the more complex algorithm presented above.

5 Conclusion

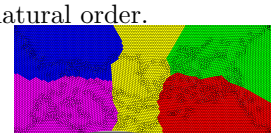
In this paper, we have presented a multilevel framework to represent target architectures in the context of process-processor placement. This representation allows the SCOTCH mapping software to compute efficiently mappings of process graphs onto irregular and possibly disconnected partitions of very large parallel machines. This removes an important bottleneck of such software, in order to address the process mapping needs of petascale and exascale applications.



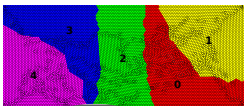

```
sub 5 0 4 1 5 7
mesh2D 4 2
```

a. Description of a 5-vertex part of a 2×4 grid, with vertices labeled in a different order than the natural order.

b. The `bump` graph.



c. Partitioning into 5 parts.

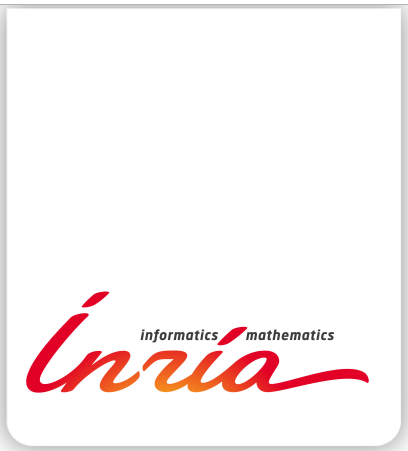


d. Mapping onto the target architecture of Figure 8.a.

Figure 8: Mapping of a graph onto a renumbered part of a 2D-grid of 2×4 vertices, compared to plain partitioning.

Acknowledgments

This work is integrated and supported by the ELCI project, a French FSN ("*Fonds pour la Société Numérique*") project that associates academic and industrial partners to design and provide software environment for very high performance computing.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399