UNIVERSITÉ DE BORDEAUX I
LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE

HABILITATION À DIRIGER DES RECHERCHES

AU TITRE DE L'ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **François PELLEGRINI**

Contributions au partitionnement de graphes parallèle multi-niveaux

(Contributions to parallel multilevel graph partitioning)

Soutenue et présentée publiquement le : 3 décembre 2009

Après avis des rapporteurs :

Frédéric DESPREZ	Directeur de recherche, INRIA
Bruce HENDRICKSON	Senior scientist, SANDIA
Burkhard MONIEN ...	Prof. Dr., Universität Paderborn

Devant la commission d'examen composée de :

Rémi ABGRALL	Professeur, IPB	Examinateur
Olivier BEAUMONT ..	Directeur de recherche, INRIA	Examinateur
Michel COSNARD	Professeur, U. Nice-Sophia Antipolis PDG, INRIA	Examinateur
Frédéric DESPREZ	Directeur de recherche, INRIA	Rapporteur
Cyril GAVOILLE	Professeur, U. Bordeaux 1 & IUF ..	Examinateur
Bruce HENDRICKSON	Senior scientist, SANDIA	Rapporteur

– 2009 –

Acknowledgements

This *habilitation* dissertation presents the current state of work on parallel graph partitioning methods which I have been carrying out for more than a decade. It is a great pleasure for me to reserve this page as a mark of gratitude to all those who participated in the realization of this milestone.

First, I wish to thank Michel Cosnard for having taken the time, in a most busy schedule, to come to Bordeaux to preside over this jury. Michel was also a member of my PhD jury, fourteen years ago, and I have been harassing him for years to be on this one. I am glad he could do it. I hope we will be able to talk shop from time to time, as we did during the famous Euro-Par'2006 Dresden banquet, where we conjectured that German waiters, trained to serve tables of eight seats, could take into account a ninth surprise guest in three iterations of dish service.

Many thanks also to Bruce Hendrickson for having undertaken such a long trip for just a few hours of presence here. Being one of the authors of the Chaco software, Bruce has been a long-time "coopetitor" in the graph partitioning business. His friendly interest in my work, materialized by his willingness to report on my dissertation and to participate in my jury, is very touching indeed.

Another coopetitor in the graph partitioning business on this side of the Atlantic is the team lead by Pr. Dr. Burkhard Monien. I deeply regret that, because of schedule constraints, Pr. Monien could not be part of my jury. It is consequently all the more nice of him to have accepted to report on my work.

Frédéric Desprez would have been a local member of my jury, if years ago he had not run away (and cycle, and swim, too), back to Lyon. It is with great pleasure that I welcome him as a foreigner, to close a question he opened twelve years ago. It is indeed in 1997 that he started telling his students that a parallel version of Scotch was to be expected "soon". It is often said that in the software business, like in the construction business, delays are frequent. Well, with a little delay, I am pleased to present him the said version.

I'm happy that Olivier Beaumont accepted to be part of my jury. Our busy lives did not give us many occasions to discuss the current state of our respective research on distributed graph algorithms, so that we had to resort to this most formal way to have me present it to him in detail. I hope that other, more informal, occasions will happen in the near future.

As another prominent member of the local graph theory team, Cyril Gavoille personifies both the reminiscence of my interest in graph theory, which takes its roots in my early PhD years, and the reminder that I can always put more theory in my papers. More than that, he is a most pleasant colleague, the opinion of whom on my work is warmly welcomed.

Last but not least, Rémi Abgrall is not only an esteemed colleague, but also represents the most welcome bridge between applied mathematics and computer science in Bordeaux. Thanks

to the late ScAlApplix project, the links between these two communities have started to be tightened towards the realization of common research goals and software tools. I hope that the Bacchus project will benefit from my past and future work.

A habilitation jury is unfortunately too small to welcome all the people I would have liked to participate in the evaluation of my work, as they have contributed to its very beginning. Let me cite André Raspaud, one of my masters in graph theory, as well as Jean Roman, former head of the ScAlApplix project, and before that of the "*Parallel Programming and Environments*" team at LaBRI, who welcomed my very first work on graph partitioning algorithms, making all of the above possible.

Also, many thanks to Cédric Chevalier and Jun-Ho Her for their contributions to the materials presented in this document. It is a great pity that they could not be here to have a piece of the SCOTCH cake with me. Thanks also to all the past SCOTCH team interns, and to Sébastien Fourestier and Cédric Lachat, who are participating in writing the new page that begins right now, after turning this one.

Contents

Acknowledgements

This *habilitation* dissertation presents the current state of work on parallel graph partitioning methods which I have been carrying out for more than a decade. It is a great pleasure for me to reserve this page as a mark of gratitude to all those who participated in the realization of this milestone.

First, I wish to thank Michel Cosnard for having taken the time, in a most busy schedule, to come to Bordeaux to preside over this jury. Michel was also a member of my PhD jury, fourteen years ago, and I have been harassing him for years to be on this one. I am glad he could do it. I hope we will be able to talk shop from time to time, as we did during the famous Euro-Par'2006 Dresden banquet, where we conjectured that German waiters, trained to serve tables of eight seats, could take into account a ninth surprise guest in three iterations of dish service.

Many thanks also to Bruce Hendrickson for having undertaken such a long trip for just a few hours of presence here. Being one of the authors of the Chaco software, Bruce has been a long-time "coopetitor" in the graph partitioning business. His friendly interest in my work, materialized by his willingness to report on my dissertation and to participate in my jury, is very touching indeed.

Another coopetitor in the graph partitioning business on this side of the Atlantic is the team lead by Pr. Dr. Burkhard Monien. I deeply regret that, because of schedule constraints, Pr. Monien could not be part of my jury. It is consequently all the more nice of him to have accepted to report on my work.

Frédéric Desprez would have been a local member of my jury, if years ago he had not run away (and cycle, and swim, too), back to Lyon. It is with great pleasure that I welcome him as a foreigner, to close a question he opened twelve years ago. It is indeed in 1997 that he started telling his students that a parallel version of Scotch was to be expected "soon". It is often said that in the software business, like in the construction business, delays are frequent. Well, with a little delay, I am pleased to present him the said version.

I'm happy that Olivier Beaumont accepted to be part of my jury. Our busy lives did not give us many occasions to discuss the current state of our respective research on distributed graph algorithms, so that we had to resort to this most formal way to have me present it to him in detail. I hope that other, more informal, occasions will happen in the near future.

As another prominent member of the local graph theory team, Cyril Gavoille personifies both the reminiscence of my interest in graph theory, which takes its roots in my early PhD years, and the reminder that I can always put more theory in my papers. More than that, he is a most pleasant colleague, the opinion of whom on my work is warmly welcomed.

Last but not least, Rémi Abgrall is not only an esteemed colleague, but also represents the most welcome bridge between applied mathematics and computer science in Bordeaux. Thanks

to the late ScAlApplix project, the links between these two communities have started to be tightened towards the realization of common research goals and software tools. I hope that the Bacchus project will benefit from my past and future work.

A habilitation jury is unfortunately too small to welcome all the people I would have liked to participate in the evaluation of my work, as they have contributed to its very beginning. Let me cite André Raspaud, one of my masters in graph theory, as well as Jean Roman, former head of the ScAlApplix project, and before that of the "*Parallel Programming and Environments*" team at LaBRI, who welcomed my very first work on graph partitioning algorithms, making all of the above possible.

Also, many thanks to Cédric Chevalier and Jun-Ho Her for their contributions to the materials presented in this document. It is a great pity that they could not be here to have a piece of the SCOTCH cake with me. Thanks also to all the past SCOTCH team interns, and to Sébastien Fourestier and Cédric Lachat, who are participating in writing the new page that begins right now, after turning this one.

Chapter 1

Introduction

The work presented in this dissertation relates to the Scotch project. This project was carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I, and now within the Bacchus team[1] of INRIA Bordeaux – Sud-Ouest. This research project started in 1992, at the beginning of my second PhD year. At that time, members of our "*Parallel Programming and Environments*" team at LaBRI worked on a software environment for the CHEOPS multicomputer which was part of the team's plans [19, 20, 27, 79, 92]. Given the pyramidal shape of the interconnection network[2], a tool was required to adequately map communicating processes onto the processors of the machine, so as to minimize inter-processor communication and network congestion [92]. However, most of the algorithms which existed at that time had a complexity in time well above linear [3, 11].

This is why my first work on this subject aimed at providing a fast and efficient static mapping procedure. Because of the orientation taken during my first PhD year, working on interconnection network properties with professor Johnny Bond before he left to Nice university, the approach which I decided to follow in agreement with my other advisor, professor Jean Roman, was to consider a "divide and conquer" approach. This first work resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and in the study of several graph bipartitioning heuristics [93]. To represent the fact that two communicating processes should not be placed too far apart across processors of the target machine, one can imagine that these processes are bound by a rubber string, which should be stretched as little as possible. When doing plain graph partitioning in a recursive way, once a vertex of the original graph is placed onto some subset of the processes, all of the rubber strings connecting it to vertices in the other part are no longer considered, as if they were pinned on the boundary. What happens afterward to vertices belonging to one side of the boundary will never impact subsequent decisions for the vertex belonging to the other part. On the other hand, with static mapping, the tension produced by moving a vertex further apart from the boundary should induce that connected vertices on the other side of the boundary be placed closer to it, so as to reduce the stretch of the cross-boundary strings. In this case, strings are not considered to be pinned on the boundary, but instead are stuck by bridges of adhesive tape which allow them to slide and propagate tension constraints. The name Scotch was coined by reference to this analogy[3].

By the time I completed this work [99], Jean Roman started supervising another PhD on parallel sparse linear solvers. My next task was therefore to transpose the existing recursive

[1] One of the two offsprings of the late ScAlApplix project [106].

[2] Hence the name of the machine.

[3] Another possible reason which is also cited is the fondness of the author for old single malt whiskies, especially of Islay and of the Highlands.

edge bipartitioning scheme to vertex separators, so as to be able to compute high-quality vertex separators for the ordering of sparse matrices by nested dissection [100, 101]. In a few years, a complete software chain for solving structural mechanics problems was designed and implemented [47, 48], under contract from the CESTA center of the French CEA (atomic energy agency). Its scientists, who had access to very large machines, began generating problems which were so large that the sequential ordering phase soon became the main bottleneck because these large graphs did not fit in memory and resulted in heavy disk swapping. In order to overcome this difficulty, a first solution was to extend the ordering capabilities of SCOTCH to handle native mesh structures, in the form of bipartite graphs. These node-element bipartite graphs are a representation of hypergraphs, for which vertex separation algorithms were designed [98]. For the type of elements which were considered, memory usage was divided by a factor of three, at the expense of an equivalent increase in run time. Yet, in the meantime, problem size went on increasing, so that the problem remained.

For these reasons, the PT-SCOTCH project (for "*Parallel Threaded* SCOTCH") was started [109]. Several parallel graph partitioning tools had already been developed by other teams in the meantime [62, 83], but serious concerns existed about their scalability, both in terms of running time and of the quality of the results produced. In order to anticipate the expected growth in problem and machine sizes, our design goal was to be able to partition graphs of above one billion vertices, distributed over a thousand processors. Because of this deliberately ambitious goal, scalability issues had to receive considerable attention.

This dissertation presents the main results which we obtained in our attempt to fulfill these objectives.

Chapter 2

Context

The conceptual framework in which my research takes place is at the junction of two widely studied fields. The first one is parallel computing, more specifically high performance scientific computing. The second one is graph theory, as graphs are a most powerful model to represent interrelated entities. Based on the joint use of the tools they provide has gathered a small but lively community which refers to itself as *combinatorial scientific computing*, which is "*concerned with the formulation, application and analysis of discrete methods in scientific applications*" [51, 55].

Benefiting from this mindset and toolset, the work presented in this dissertation focuses on two problems which are critical for our research team: sparse matrix ordering and graph partitioning.

2.1 Elements of graph theory

Readers interested in a thorough review on graph theory may refer to [10].

Graphs

An *unoriented graph* $G(V, E)$ is a structure made of a set V of elements called *vertices*, which we will assume of finite size in all of the following, and of a set E of pairs of vertices called *edges*. The vertex and edge sets of graph G may be referred to as $V(G)$ and $E(G)$, respectively. The number of vertices of a graph is called the *order* of the graph. It is also commonly referred to as its *size*.

When graph vertices are weighted, $w(v)$ denotes the vertex weight of some vertex v. When graph edges are weighted, $w(e)$ denotes the weight of some edge e. In all of the following, vertex and edge weights are assumed to be integer and strictly greater than zero.

An edge $\{v', v''\}$ is said to be *incident* to v' and to v''. v' and v'' are the *ends* of $\{v', v''\}$, and are said to be *adjacent*. An edge of the form $\{v, v\}$ is called a *loop*. An edge existing as multiple occurrences in the edge set is a *multiple edge*.

A *simple* graph is a graph without any loop nor any multiple edges. In all of the following, we will only consider simple graphs.

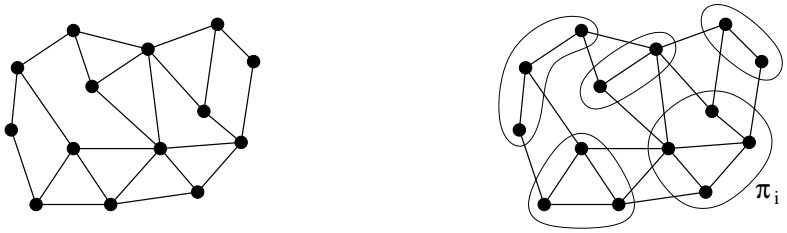
Let v be a vertex of a graph G. The *degree* of v, noted $\delta(v)$, is the number of edges of $E(G)$ incident to v. This notion is extended to the whole graph: the *minimum degree* of G, $\delta(G)$, is the minimum of $\delta(v)$ over all vertices v of $V(G)$. Similarly, the *maximum degree* of G, noted $\Delta(G)$, is the maximum of $\delta(v)$ over all vertices v of $V(G)$.

Let V' be a subset of $V(G)$. The *cocycle* $\omega(V')$ of V' is the set of edges having exactly one of their ends in V', that is, $\omega(V') \stackrel{\text{def}}{=} \{v : \exists (v, v') \in E(G)/v' \in V', v \notin V'\}$.
Consequently, for any vertex v of $V(G)$, $\omega(\{v\})$ represents the neighborhood of v, that is, the set of all of the vertices adjacent to it.
A *path* between two vertices v' and v'' of a graph G is a list $\{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\}$ of edges of $E(G)$ such that $v' = v_1$ and $v'' = v_k$. The number of edges in a path is called the *length* of the path. The *distance* between two vertices v' and v'' of a graph G is the length of the smallest path existing between v' and v''. It is infinite if no such path exists.

A graph G is said to be *connected* if a path exists between every pair of vertices in the graph. The *diameter* of a connected graph G, noted $\text{diam}(G)$, is the maximum, over all edges (v', v'') of G, of the distance between v' and v''.
A vertex v' of a connected graph G is *peripheral* if a vertex v'' of G exists such that the distance between v' and v'' is equal to $\text{diam}(G)$.

Partitions and quotient graphs

Let $V(G)$ be the non-empty vertex set of a graph G of order n. A *partition* Π of $V(G)$ is the splitting of $V(G)$ into N vertex subsets π_i, with $1 \leq i \leq N$, called parts, such that: (i) no part π_i is empty: $\forall i, \pi_i \neq \emptyset$; (ii) all parts are pairwise disjoint: $\forall i, j/i \neq j, \pi_i \cap \pi_j = \emptyset$; (iii) the union of all parts is equal to $V(G)$: $\cup_i \pi_i = V(G)$. An example of partition is given in Figure 2.1.



(a) Some graph G.

(b) A partition Π of G with some part π_i.

Figure 2.1: Partition of a graph.

For some partition Π and for each vertex v of G, $\pi(v)$ is the part of Π which contains v.
By extension, $\Pi(V')$, where V' is a subset of $V(G)$, represents the set of parts which contain all vertices of V': $\Pi(V') \stackrel{\text{def}}{=} \{\pi_i \in \Pi : \exists v' \in V'/\pi(v') = \pi_i\}$.

The *cut* of a partition Π is the union of the cocycles of every part: $\omega(\Pi) \stackrel{\text{def}}{=} \cup_{\pi_i \in \Pi} \omega(\pi_i)$.

The *halo* $\lambda(V')$ of some subset V' of $V(G)$ is the set of vertices which are the ends of the cocycle edges of V' which do not belong to V' itself: $\lambda(V') \stackrel{\text{def}}{=} \{v : \{v, v'\} \in \omega(V') \text{ and } v \notin V'\}$.

A part is said to be adjacent to another if at least one of its vertices belongs to the halo of the other part.

The *quotient graph* $Q = G_{/\Pi}$ of some graph G with respect to some partition Π of $V(G)$ is defined by:

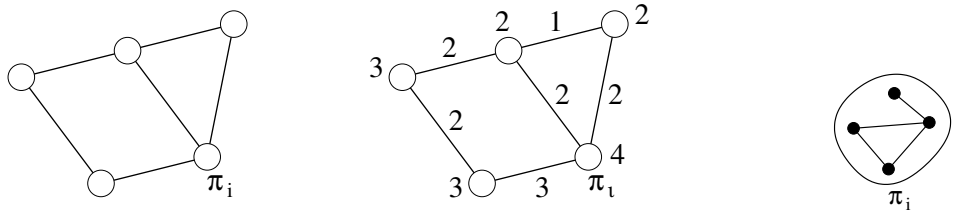
$$\begin{cases} V(Q) = \Pi \ , \\ (\{\pi_i, \pi_j\} \in E(Q)) \iff (i \neq j,\ \exists v' \in \pi_i,\ \exists v'' \in \pi_j \ / \ \{v', v''\} \in E(G)) \ . \end{cases}$$

When weighted quotient graphs are considered, their vertex weights are equal to the sum of the weights of the coalesced vertices, and their edge weights are equal to the sum of the weights of the original edges that they replace, as illustrated in Figure 2.2.b:

$$w(\pi_i) = \sum_{v \in \pi_i} w(v) \ ,$$

$$w(\{\pi_i, \pi_j\}) = \sum_{\substack{v' \in \pi_i, v'' \in \pi_j \\ \{v', v''\} \in E(G)}} w(\{v', v''\}) \ , \text{ with } i \neq j.$$

For every part π of Π, we denote $G[\pi]$ the *subgraph* of G *induced* by π, that is, the graph the vertex set of which is π, and the vertex set of which is the subset of $E(G)$ all edges of which have both of their ends in π.



(a) Unweighted graph $G_{/\Pi}$. (b) Weighted graph $G_{/\Pi}$. (c) Graph $G[\pi_i]$.

Figure 2.2: Quotient and induced graphs resulting from partition Π of graph G, both represented in Figure 2.1.

Mappings

The *mapping* of a graph G to a graph H is a couple $\xi_{G,H} = (\tau_{G,H}, \rho_{G,H})$, where $\tau_{G,H}$ is an injective application of $V(G)$ to $V(H)$, and $\rho_{G,H}$ is an injective application which associates, with every edge $\{v', v''\}$ of G, a path $\rho_{G,H}(\{v', v''\})$ of H linking $\tau_{G,H}(v')$ to $\tau_{G,H}(v'')$.

The *dilation* of a mapping $\xi_{G,H} = (\tau_{G,H}, \rho_{G,H})$ is defined by:

$$\mathrm{dil}(\xi_{G,H}) \stackrel{\text{def}}{=} \max_{\{v', v''\} \in E(G)} (|\rho_{G,H}(\{v', v''\})|) \ ,$$

where $|\rho_{G,H}(\{v', v''\})|$ represents the length of path $\rho_{G,H}(\{v', v''\})$. This is the length of the longest path induced by the mapping.

The *congestion* of a mapping $\xi_{G,H} = (\tau_{G,H}, \rho_{G,H})$ is:

$$\mathrm{cg}(\xi_{G,H}) \stackrel{\text{def}}{=} \max_{\{w',w''\}\in E(H)} (|\{\rho_{G,H}(\{v',v''\}) \,/\, \{v',v''\} \in E(G) \text{ and } \{w',w''\} \in \rho_{G,H}(\{v',v''\})\}|) \ ,$$

where $\{w', w''\} \in \rho_{G,H}(\{v', v''\})$ indicates that path $\rho_{G,H}(\{v', v''\})$ uses edge $\{w', w''\}$ of H. It is the maximum number, for every edge of H, of the number of mapping paths which use this edge.

2.2 Elements of sparse linear algebra

2.2.1 Matrices

A *matrix* is a rectangular array of elements, or entries, set out by rows and columns. A matrix A with m rows and n columns is denoted by $A(m, n)$. Element $a_{i,j}$ of matrix A is the element located at row i, column j.
A *square* matrix is a matrix which has the same number of rows and columns. The *order* n of a square matrix $A(n, n)$ represents its number of rows and columns.

In all of the following, we will deal with square matrices the entries of which are scalar values, either integer, real or complex.

A matrix is *symmetric* if, for all indices i and j, $a_{i,j} = a_{j,i}$. It is said to be *structurally symmetric*, or to have a *symmetric pattern*, if and only if, for all indices i and j, $(a_{i,j} \neq 0) \iff (a_{j,i} \neq 0)$.

A symmetric matrix of order n is called *positive-definite* if, for all nonzero vectors $x \in \mathbb{R}^n$, the associated quadratic form given by $Q(x) = x^T Ax$ takes only strictly positive values.

A matrix is said to be *sparse* if most of its entries are zeros. From a practical point of view, a matrix is said to be sparse if it contains enough zero entries to be worth taking advantage of them to reduce both the storage and work.
The *density* of a sparse matrix is the ratio of the number of its non zero entries over its number of entries. In combinatorial scientific computing, most sparse matrices have a number of non zeros entries per row and per column which is bounded by a constant depending on the type of the problem, rather than on problem size. Consequently, the density of these matrices decreases along with their order.

Let $A(n, n)$ be a square matrix of order n, either symmetric or with a symmetric pattern. The *adjacency graph* of A is the graph $G(V, E)$ with n vertices, one per row and column, such that there exists an edge (v_i, v_j) if and only if $a_{i,j}$ and $a_{j,i}$ are non zero.
Adjacency graphs are relevant only for sparse matrices, many structural properties of which can be reflected as topological graph properties.

2.2.2 Linear systems

A *system of linear equations*, or *linear system*, is a collection of linear equations involving the same set of variables. It can be represented under the matrix form $Ax = b$, where $A(m, n)$ is the matrix of coefficients, with m rows (the equations) and n columns (the unknowns), x is a column vector with n entries, and b is a column vector with m entries.
A *sparse linear system* is a system of linear equations the coefficient matrix of which is sparse.

Gaussian elimination is a method of solving a system of n linear equations in n unknowns, in which there are first $(n-1)$ steps, the m^{th} step of which consists in subtracting a multiple of the m^{th} equation from each of the following ones so as to eliminate one variable, resulting in a triangular set of equations. This triangular system can then be solved by back substitution, computing the n^{th} variable from the n^{th} equation, the $(n-1)^{\text{th}}$ variable from the $(n-1)^{\text{th}}$ equation, and so on.

The *fill-in* of a matrix represents its entries which change from an initial zero to a non-zero value during the execution of an algorithm such as Gaussian elimination.
By nature of Gaussian elimination, a zero $a_{i,j}$ term of a matrix will incur fill-in during factorization if there exists in the associated adjacency graph a path linking vertices v_i and v_j such that all of its intermediate vertices have indices smaller than $\min(i,j)$ [1].

An *ordering* φ of the vertices of some graph G is a bijection between $V(G)$ and $\{0,1,\ldots,n-1\}$.

A *permutation vector* $P(n)$ of size n is a vector which contains only once every integer in the range $\{0,1,\ldots,(n-1)\}$. If i is some index in $[0;(n-1)]$, $P(i)$ is the application of (the direct) permutation P to i, while $j/P(j)=i$ is the result of the application of the inverse permutation P^{-1} to i. Hence, $\forall i \in [0;(n-1)], P(P^{-1}(i)) = P^{-1}(P(i)) = i$.

A *permutation matrix* $P(n,n)$ is a square $(0,1)$-matrix that has exactly one entry 1 in each row and each column, and 0's elsewhere. Each such matrix represents a specific permutation of n elements and, when used to multiply another matrix, can produce that permutation in the rows or columns of the other matrix.

A *symmetric reordering* is a pre-processing step applied to a sparse linear system prior to its factorization. This reordering is chosen in such a way that pivoting down the diagonal in order on the resulting permuted matrix PAP^T produces much less fill-in and work than computing the factors of A by pivoting down the diagonal in the original order.

2.3 Sparse matrix reordering

Many scientific and engineering problems can be modeled by sparse linear systems, which are solved either by iterative or direct methods. Direct methods are popular because of their generality and robustness, especially in some application fields such as linear programming, structural mechanics or magneto-hydrodynamics, where iterative methods may fail to achieve convergence for specific classes of problems. However, to solve efficiently large sparse linear systems with direct methods, one must minimize the amount of fill-in induced by the factorization process.

As described above, fill-in is a direct consequence of the order in which the unknowns of the linear system are numbered, and its effects are critical both in terms of memory and computation costs. In the general case, great care should be exercised when permuting rows and column of matrices in the course of Gaussian elimination, because of the potential numerical instabilities that could result from the amplification of rounding errors. However, for symmetric definite-positive matrices, the LL^T Cholesky factorization that is used in lieu of the traditional LU factorization exhibits higher numerical stability, which is independent of the order in which the

[1] It may happen that the new (i,j) coefficient is numerically equal to zero. Yet, in all sparse linear system solving software, some space is pre-allocated to store such coefficients anyway. We will refer to the latter case as *symbolic* fill-in, as opposed to *numerical* fill-in.

unknowns are considered [46]. Consequently, the two problems of reordering and numerical factorization can be safely decoupled.

2.3.1 Nested dissection

An efficient way to compute fill reducing orderings of symmetric sparse matrices is to use recursive nested dissection [43]. It amounts to computing a vertex set S which separates the adjacency graph associated with some matrix into two parts A and B, ordering S last, and proceeding recursively on parts A and B until their sizes become smaller than some threshold value. This ordering guarantees that no non-zero term can appear in the factorization process between unknowns of A and unknowns of B.

The main issue of the nested dissection ordering algorithm is thus to find small vertex separators which balance the remaining subgraphs as evenly as possible. Vertex separators are commonly computed by means of vertex-based graph algorithms [56, 78], or from edge separators [102, and included references] using minimum cover techniques [33, 60], but other techniques such as spectral vertex partitioning have also been used [103]. Many theoretical results have been carried out on the nested dissection ordering [18, 80], and its divide and conquer nature makes its parallelization easy at a coarse-grain level[2]. Provided that good vertex separators are found, the nested dissection algorithm produces orderings which, both in terms of fill-in and operation count, compare very favorably [49, 64, 100] to those based on the minimum degree algorithm.

The minimum degree algorithm [114] is a local heuristic which is very often efficient and extremely fast, thanks to the many improvements which have been made to it [1, 43, 44, 81], but it is intrinsically sequential, so that attempts to derive parallel versions of it have not been successful [21], especially for distributed-memory architectures.

Moreover, the elimination trees induced by nested dissection are broader, shorter, and more balanced than those from multiple minimum degree [100], and therefore exhibit much more concurrency in the elimination tree, which is essential in order to achieve high performance when factorizing and solving these linear systems on parallel architectures [6, 41, 42, 49, 108].

2.3.2 Ordering quality metrics

The quality of orderings is commonly evaluated with respect to two criteria. The first one, NNZ ("number of non-zeros"), is the number of non-zero terms in the factorized reordered matrix, which can be divided by the number of non-zero terms in the initial matrix to give the fill ratio of the ordering. The second one, OPC ("operation count"), is the number of arithmetic operations required to factor the matrix using the Cholesky method. It is equal to $\sum_c n_c^2$, where n_c is the number of non-zeros of column c of the factorized matrix, diagonal included.

NNZ and OPC are assumed to be indirect measurements of the overall quality of the nested dissection process, that is, of the quality of the separators produced at each of its stages. Since the diagonal blocks associated with separator vertices are almost always full in the factorized matrix[3], small separators are likely to result in smaller such blocks, with fewer extra-diagonal connections, that is, less fill-in. Yet, this is not always the case, as many hazards can impact the fill-in of the matrix, and consequently the number of operations to perform.

[2] Once a separator has been computed on a graph distributed on p processes, each of the separated parts can be redistributed onto $\frac{p}{2}$ processes, such that the nested dissection algorithm can recursively operate independently on each of the process subsets, see *e.g.* [24].

[3] This property derives from the fact that, between any two vertices of some separator, a path always exists linking these two vertices such that all path vertex indices are smaller than the minimum of the indices of the two ends. If it is not the case, either the graph is disconnected or one of the end vertices can be safely removed from the separator.

2.4 Graph partitioning

Graph partitioning is a ubiquitous technique which has applications in many fields of computer science and engineering. It is mostly used to help solving domain-dependent optimization problems modeled in terms of weighted or unweighted graphs, where finding good solutions amounts to computing, possibly recursively in a divide-and-conquer framework, small vertex or edge cuts that evenly balance the weights of the graph parts.

Since the computation of a balanced partition with minimal cut is NP-hard, even in the bipartitioning case [39, 40], many heuristics have been proposed to provide acceptable partitions in reasonable time. It is also the case for some of its variants, such as the mapping problem which will be described below.

2.4.1 Data distribution for parallel processing

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 entire duration of the program, this optimization problem is referred to as *mapping*. It amounts to balancing the computational weight of the processes among the processors of the machine, while reducing the cost of communication by keeping intensively inter-communicating processes on nearby processors.

In most cases, the underlying computational structure of the parallel programs to map 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[4] context, this is equivalent to the *distribution* across processors of the data structures of parallel programs; in this case, all pieces of data assigned to some processor are handled by a single process located on this processor. This is why, in all of the following, we will talk about *processes* rather than *processors* or *processing elements* to represent the entities across which data will be distributed.

When the architecture of the target machine is assumed to be fully homogeneous in term of communication, this problem is called the *partitioning* problem, which has been intensely studied [8, 53, 64, 65, 103]. Graph partitioning can be seen as a subproblem of mapping where the target architecture is modeled as a complete graph with identical edge weights, while vertex weights can differ according to the compute power of the processing elements running the processes of the SPMD program. On the other hand, mapping can be seen as the inclusion of additional constraints to the standard graph partitioning problem, turning it into a more general optimization problem termed *skewed graph partitioning* by some authors [54].

2.4.2 Mapping quality metrics

The computation of efficient static mappings or graph partitions 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.

While in some methods load imbalance and communication minimization are placed at the same priority level, being aggregated in a unique cost function by means of weighting

[4] *Single Process, Multiple Data.* This is a programming paradigm in which all processes of a parallel program possess the very same code, the individual behavior of the processes being conditioned by the data they handle.

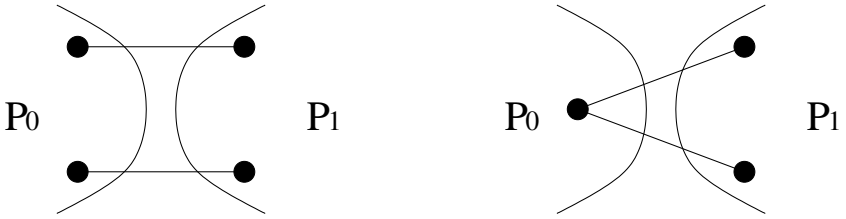


Figure 2.3: Inaccuracy of the graph partitioning model for representing data exchange. For both of the above bipartitions, $f_C = 2$, but 4 vertex data have to be exchanged in the leftmost case, while it is 3 in the rightmost case.

coefficients[5], load imbalance takes precedence in most cases: the goal of the algorithms is then to minimize some communication cost function, while keeping load balance within a specified tolerance.

One of the most widely used cost functions for communication minimization is the sum, for all edges, of their dilation multiplied by their weight [34, 50, 53, 93]:

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

In the case of graph partitioning, where all edge dilations are equal to 1, this function simplifies into:

$$f_C(\Pi) \stackrel{\text{def}}{=} \sum_{e \in \omega(\Pi)} w(e) \ .$$

The f_C function has several interesting properties. First, it is easy to compute, and it allows for incremental updates performed by iterative algorithms. Second, 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, thus the risk of congestion, and its minimization favors the mapping of intensively intercommunicating processes onto nearby processors [50, 93, 123]. However, it may be inaccurate to model data exchange for parallel computations. For instance, in the case of hybrid[6] or iterative solving of sparse linear systems, the main operation to perform is a parallel matrix-vector product. Data borne by frontier vertices have to be sent to neighboring processes, but only once per destination process, while the edge cut may account for several edges incident to the same vertex, as illustrated in Figure 2.3. This is why many authors have advocated for a hypergraph-based communication cost model, and use hypergraph partitioning tools instead of graph partitioning tools to assign vertices to subdomains [14, 15, 52].

Although hypergraph partitioning represents data exchange in an exact way, yielding an average decrease of communication cost of 5 to 10 percent on mesh graphs, and up to 60 percent for some non-mesh graphs, compared to plain graph partitioning [14], it is several times slower than the latter, which is therefore still of economic interest for scientific computing. In particular, for 2D or 3D mesh graphs of bounded degree, which result in smooth and continuous

[5]This is for instance the case for genetic algorithms [72, 105], where individuals are evaluated by means of a single cost function (see Section 4.4). Although this function could be crafted such that imbalanced partitions yield infinite cost, so as to privilege balance enforcement over communication minimization, this is not desirable as plateau effects in the cost function hinder the convergence of these algorithms.

[6]Hybrid solving is a paradigm which tries to combine the stability of direct methods and the speed and small memory footprint of iterative methods for solving large sparse systems of equations. For instance, some hybrid methods use a Schur complement decomposition to perform direct linear system solving on the separated subdomains distributed across the processes, and to perform iterative solving on the separators, as in [38].

separators, the effective difference between the graph and hypergraph cost models is reduced for interface faces, with respect to a constant factor which depends on the type and average degree of the meshes.

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

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

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

δ_{map} represents the normalized average, across all target vertices, of the absolute difference between the actual load per computational power unit and μ_{map}, the ideal average load per computational power unit.

2.5 Multilevel framework

Experience has shown that, for many graphs used in scientific computations[7], best partition quality is achieved when using a multilevel framework. This method, which derives from the multigrid algorithms originally used in numerical physics, repeatedly reduces the size of the graph to partition by clustering vertices and edges, computes an initial partition for the coarsest graph obtained, and prolongs[8] the result back to the original graph [8, 53, 67], as illustrated in Figure 2.4.

2.5.1 Refinement

Multilevel methods are most often combined with partition refinement methods, in order to smooth the prolonged partitions at every level so that the granularity of the solution is that of the original graph and not that of the coarsest graph.

In the sequential case, the most popular refinement heuristics in the literature are the local optimization algorithms of Kernighan-Lin [73] (KL) and Fiduccia-Mattheyses [37] (FM), either on edge-based [53] or vertex-based [56] forms depending on the nature of the desired separators. All of these algorithms are based on successive moves of frontier vertices to decrease the current value of the prescribed cost function. Moves which adversely affect the cost function may be accepted, if they are further compensated by moves which result in an overall gain, therefore allowing these algorithms to perform *hill-climbing* from local minima of the cost function. In [30], groups of judiciously selected vertices (called *helpful sets*) are moved together[9], while

[7] These graphs have in common good locality and separation properties: their maximum degree is small, and their diameter evolves as a fractional power of their size, *e.g.* in $O\left(n^{\frac{1}{d}}\right)$ for d-dimensional mesh graphs of size n.

[8] While a *projection* is an application to a space of lower dimension, a *prolongation* refers to an application to a space of higher dimension. Yet, the term projection is also used to refer to the propagation of the values borne by coarse graph vertices to the finer vertices they represent.

[9] Such collective moves allow for partial hill-climbing of the cost function outside of the traditional operations of the KL and FM algorithms, since a helpful set can contain vertices of both negative and positive gains.

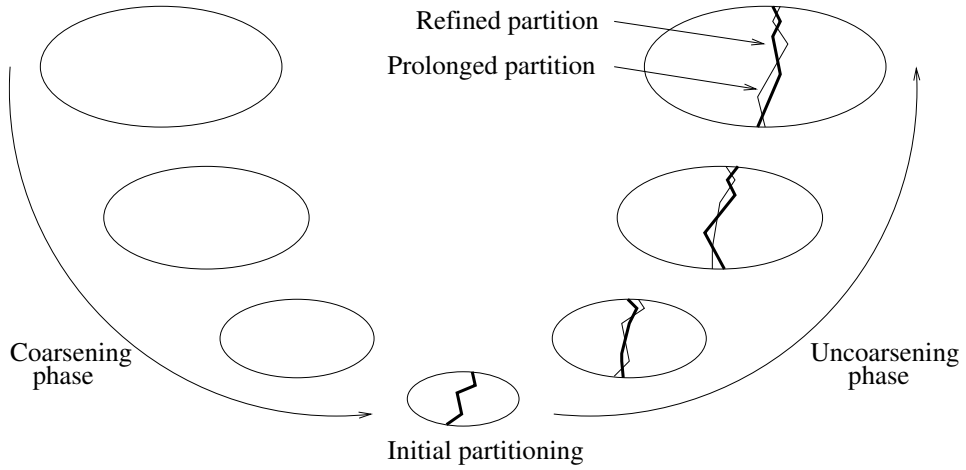
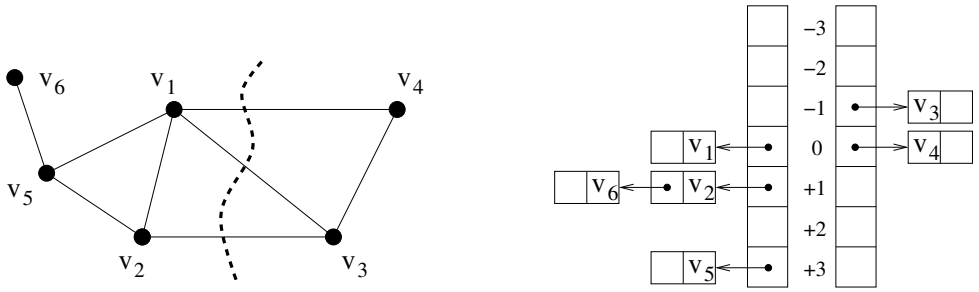


Figure 2.4: The multilevel partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the prolonged partition obtained from the coarser graph, and the partition obtained after refinement, respectively.



(a) Current bipartition of an unweighted graph.

(b) Lists and gain bucket arrays corresponding to the two parts of this bipartition.

Figure 2.5: Typical bucket data structures used by Fiduccia-Mattheyses-like algorithms to record vertex gains.

traditional implementations of KL and FM swap pairs of vertices or perform individual moves, respectively.

Vertices to be moved are chosen according to their *gain value*, denoted by ∇f_C, which is the amount by which the current value of the cost function would be modified if the given vertex were moved to some other part; vertices of negative gain values are consequently the most interesting to move. For that purpose, vertex gain values are computed in advance, and vertices of interest are inserted into data structures which implement some flavor of sorting, such that vertices with lowest gain values can be retrieved at small cost. Such data structures can be linearly [53] or logarithmically [94] indexed bucket arrays (see Figure 2.5), or trees [118]. Additional selection criteria may be implemented into these data structures, such as secondary sorting by vertex weight, so as to reduce imbalance as much as possible, by moving first the heaviest possible vertices from the most heavily loaded part to the lightest one [118].

In the case of bipartitioning, there is only one part to move the vertex to, and candidate vertices can be stored in only one such data structure when individual vertex movements are

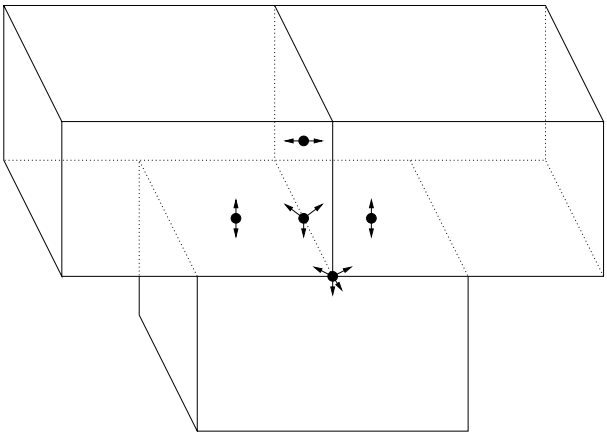


Figure 2.6: Possible moves between neighboring parts for frontier vertices in some 3D space divided into block subdomains. Vertices belonging to subdomain interface faces only have two choices, while interface edges and points have more.

considered (for FM-like algorithms) [94], or two twin structures when vertex swaps are considered (for KL-like algorithms) [65]. In the case of k-way partitioning, while most frontier vertices also have only one neighbor part (*e.g.*, the vertices belonging to subdomain interface faces for 3D meshes), others may have more (*e.g.*, vertices representing subdomain interface edges and points in 3D, see Figure 2.6).

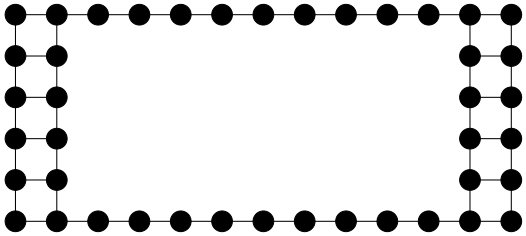
Handling frontier vertices with multiple potential destination parts requires either the use of enhanced data structures which can record several gains per vertex, that is, one per target subdomain, or some pre-selection of the most appropriate destination part if simpler data structures are sought [69].

While, in theory, all of the $(k-1)$ parts other than the current part of any vertex might be potential destinations for it, the set of destinations considered by the algorithms is always much more reduced. Since the refinement process is expected to be local by the very nature of the multilevel framework, the only vertices for which gains are computed are frontier vertices, an optimization often referred to as *boundary refinement* in the literature [53, 68]. Also, the only parts to be considered for destination of a frontier vertex are, in the context of graph partitioning, the ones which are currently adjacent to it[10] [69], which coerces the local optimization algorithm to behave in the same way as a diffusion-like method. For static mapping, the set of potential destination parts may be extended to the union of all of the parts adjacent to the part of the vertex[11] [120]. This extension with respect to the graph partitioning case comes from the potential need to introduce some "buffer" vertices between two subdomains so as to improve communication locality on the target machine, at the expense of subdomain connectivity, as illustrated in Figure 2.7.

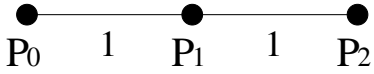
2.5.2 Coarsening

Coarsening and refinement are intimately linked. By reducing the size of large-scale, graph-wide topological structures into sets of a few tens of vertices, coarsening allows these structures to be wholly handled by local optimization algorithms, conversely broadening the scope of the latter and turning them into global-like optimization algorithms at the coarsest levels. Then,

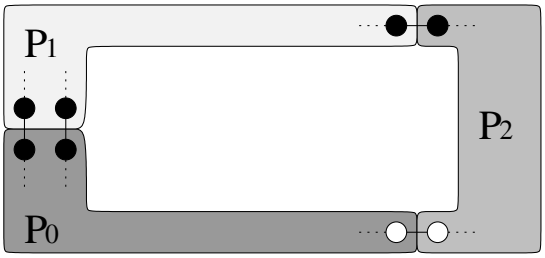
[10] That is, the set of potential destination parts of any vertex v is $\Pi(\lambda(\{v\})) \setminus \pi(v)$.
[11] That is, the set of potential destination parts of any vertex v is $\Pi(\lambda(\pi(v)))$.



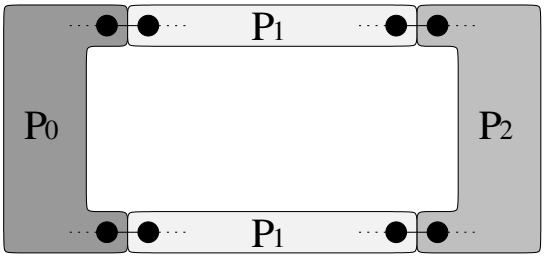
(a) A ring-shaped, unweighted graph.



(b) The target graph for static mapping. Distance between P_0 and P_1 and between P_1 and P_2 is 1, while distance between P_0 and P_2 is 2.



(c) An initial mapping, obtained for instance by a greedy graph growing algorithm. $f_C = 5$, because the cut edge between white vertices has dilation 2.



(d) Static mapping after refinement. $f_C = 4$.

Figure 2.7: Example of the interest of considering all parts adjacent to the part of the vertex when swapping vertices. Because white vertices can also be moved to P_1, which is a neighbor of P_0 as well as of P_2, buffer vertices mapped to P_1 can be inserted between vertices of P_0 and P_2, improving communication locality.

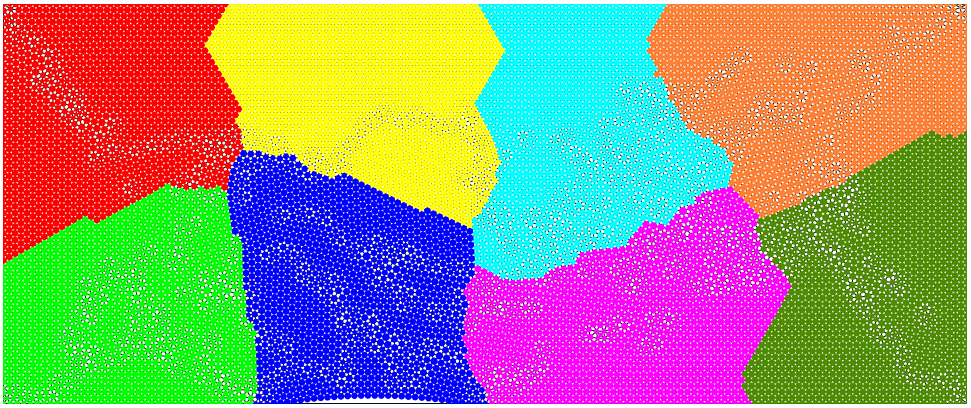


Figure 2.8: Partition of graph BUMP into 8 parts with SCOTCH 4.0, using the multilevel framework with a Fiduccia-Mattheyses refinement algorithm. The cut is equal to 714 edges. Segmented frontiers are clearly evidenced, compared to Figure 4.12, page 50.

as uncoarsening takes place, partition refinement only focuses on smaller and smaller details of the finer graphs, as large-scale structures regain their original sizes and become out of the scope of local optimization.

However, the quality of partitions produced by a multilevel approach may not be as good as that of long-running, global optimization algorithms. Indeed, the critical issue in multilevel methods is the preservation, by the coarsening algorithm, of the topological properties of the original graphs. Otherwise, solutions provided by the initial partitioning method would not be relevant approximations of the solutions to the finest problems to solve. While some of the topological bias induced by the coarsening can be removed by refinement methods, the local nature of the latter most often hinders their ability to perform the massive changes that would be required to alleviate any significant bias. Coarsening artifacts, as well as the topology of the original graphs, can trap local optimization algorithms into local optima of their cost function, such that refined frontiers are often made of non-optimal sets of locally optimal segments, as illustrated in Figure 2.8 (see also Figure 4.13.a, page 50).

Coarsening algorithms are based on the computation of vertex clusters in the fine graph, which define a partition from which the coarse graph will be built by quotienting. Clusters will become the vertices of the coarser graph, while edges of this latter will be created from the edges linking fine vertices belonging to different clusters, so that the weight of a coarse edge is equal to the sum of the weights of the fine edges it replaces (see Section 2.1).

Save for some promising work on multiple and weighted aggregation [25, 26] which aims at removing further the impact of coarsening artefacts, all of the existing coarsening schemes are based on edge matchings to mate the vertices of the fine graph, so that any coarse vertex contains at most two fine vertices. Matchings are an essential tool for the solving of many combinatorial scientific computing problems [55]. In the context of graph coarsening, matchings do not need to be maximal, which allows for the use of fast heuristics.

The most popular matching scheme in the literature is heavy-edge random matching [65] (HEM), which is very easy to implement and provides matchings of good quality on average [63]. It operates by considering all graph vertices in random order, and by mating each of them with one of its non already matched neighbors linked by an edge of heaviest possible weight. This scheme favors the preservation of the large-scale topological structure of graphs because edge weights increase with the desirability to match in a dimension which has not yet been explored.

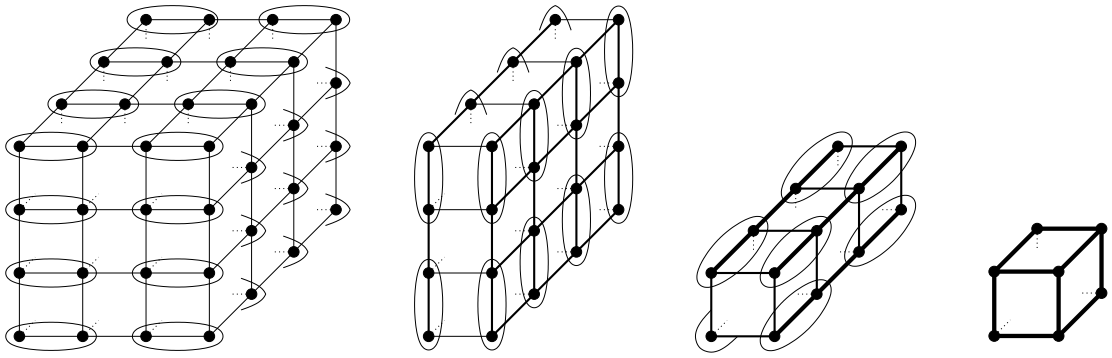


Figure 2.9: Several steps of maximal heavy edge matching and coarsening on a 3D grid, dimension after dimension. After the three dimensions have been processed, grid structure and proportions are preserved, and all edges have same weight anew.

Consider for instance a d-dimensional unweighted grid. If all vertices are matched along edges belonging to only one of the dimensions, all remaining edges in this dimension will keep their original weight, while all edges in other dimensions will be heavier. Then, if another dimension is chosen, according to the HEM policy, edges in the remaining dimension will be even heavier, and so on, such that all dimensions will be explored in turn, resulting in a grid of smaller size but identical structure, as shown in Figure 2.9. This phenomenon can also be evidenced for unstructured meshes.

In [87], Monien *et al.* propose a matching scheme based on the selection of locally heaviest edges and the post-processing of the obtained matchings by an augmenting path method. While this scheme yields matchings of heavier weights than HEM, with fewer remaining edges, it results in more imbalanced coarse graphs with respect to the distribution of the weights of coarse vertices, and induces more sequentiality in the edge selection process, which prevents its parallelization.

2.5.3 Initial partitioning

Initial partitioning algorithms are in charge of computing, on the coarsest graph, the partition which will be prolonged back to finer graphs. Since initial partition quality is mandatory, expensive algorithms may be used, all the more that they operate on small graphs.

Many global methods have been investigated to compute graph partitions from scratch, either as is or within a multilevel framework: evolutionary algorithms (comprising simulated annealing [12, 75], genetic algorithms [13, 59], ant colonies [77], greedy iterative algorithms [9, 76]), graph algorithms [35, 65], geometry-based inertial methods [89], spectral methods [8], region growing [31], etc. While taking their inspiration from radically different fields such as genetics or statistical physics, analogies between these methods are numerous, as well as cross-fertilization in their implementations, so that it is sometimes difficult to categorize them unambiguously.

Region growing algorithms have received much attention, as they result in connected parts, which is a most desirable feature in numerous application domains [31, 116]. They consist in selecting as many seed vertices in the graph as the number of desired parts, and grow regions from these seeds until region boundaries meet.

In the most basic versions of this approach, such as Farhat's algorithm [35] or the *greedy graph growing* method [65], seeds are processed one by one, and parts are built one after the

other by breadth-first traversal of the graph induced by the remaining unassigned vertices. Since this approach is not likely to produce parts of equally good shape, especially for the last ones, this algorithm is repeated several times on randomly chosen seeds, and the best partition is kept.

In more advanced algorithms, such as *bubble-growing* algorithms, parts are considered simultaneously in order to yield interfaces of higher quality [31, 85]. This class of algorithms is based on the observation that sets of soap bubbles self-organize so as to minimize the surface of their interfaces, which is indeed what is expected from a partitioning algorithm. Consequently, the idea is to grow, from as many seed vertices as the desired number of parts, a collection of expanding bubbles, by performing breadth-first traversals rooted at these seed vertices. Once every graph vertex has been assigned to some part, each part computes its center based on the graph distance metric. These center vertices are taken as new seeds and the expansion process is started again, until it converges, that is, until centers of subdomains no longer move. An important drawback of this method is that it does not guarantee that all parts will hold the same number of vertices, which requires a call to other heuristics afterward, to perform load balancing.

In [124], Wan *et al.* explore a diffusive model, called the *influence model*, where vertices impact their neighbors by diffusing them information on their current state. This model also does not handle load balancing properly.

We will not discuss further the issue of initial partitioning, as it is not directly related to our works, since this part of the multilevel framework will not be impacted by the parallelization of the latter. Indeed, as we will see in the following, once parallel multilevel frameworks coarsen distributed graphs down to some small enough size, coarsest distributed graphs can be centralized on some of the processes where traditional, sequential initial partitioning algorithms can be used. No specific improvement is therefore required.

2.6 Experimental framework

The need to process, in reasonable time, graphs of ever increasing sizes, renders the parallelization of graph partitioning unavoidable. Leaving initial partitioning aside, the contributions presented in this dissertation are aimed at devising scalable algorithms for parallel coarsening and parallel refinement, in order to provide an efficient solution to the parallel multilevel graph partitioning problem. These contributions will be described in detail in the next two chapters.

2.6.1 Basic assumptions

Our approach of the parallelization of coarsening and refinement bases on two main assumptions, regarding the nature of the parallel machine to be used, and the type of graphs to be handled.

First, we considered a distributed memory model, since large machines with hundred thousands of processing elements are not likely to ever implement efficiently a shared-memory paradigm, even of NUMA[12] type. The distributed memory paradigm has a strong impact on algorithms, because mutual exclusion and locking issues, which are implicitly resolved by doing memory accesses in the sequential case, have to be reformulated in terms of non-deterministic, latency-inducing message exchanges, both for matching and refinement.

[12]Non Uniform Memory Access. The machines implementing this model are categorized by the fact that, while their memory can be accessed from all of their processing elements, the cost of access may differ depending on the location of the memory with respect to the processing element which makes the request.

Second, we assume that distributed graphs are of reasonably small degree, that is, that graph adjacency matrices have sparse rows and columns. Unlike more robust approaches [117], we presume that vertex adjacencies can be stored locally on every process, without incurring too much memory imbalance or even memory shortage.

2.6.2 Distributed graph structure

According to the above assumptions, in our PT-Scotch library, like in other packages, distributed graphs are represented by means of adjacency lists. Vertices are distributed across processes along with their adjacency lists and with some duplicated global data, as illustrated in Figure 2.10. In order to allow users to create and destroy vertices without needing any global renumbering, every process is assigned a user-defined range of global vertex indices. Range arrays are duplicated across all processes in order to allow each of them to determine the owner process of any non-local vertex by dichotomy search, whenever necessary.

Since many algorithms require that local data be attached to every vertex, and since using global indices extensively would be too expensive in our opinion[13], all vertices owned by some process are also assigned local indices, suitable for the indexing of compact local data arrays. This local indexing is extended so as to encompass all non-local vertices which are neighbors of local vertices, which are referred to as *ghost* or *halo* vertices. Ghost vertices are numbered by ascending process number and by ascending global number, such that, when vertex data have to be exchanged between neighboring processes, they can be agglomerated in a cache-friendly way on the sending side, by sequential in-order traversal of the data array, and be received in place in the ghost data arrays on the receiving side.

A low-level halo exchange routine is provided by PT-Scotch, to diffuse data borne by local vertices to the ghost copies possessed by all of their neighboring processes. This low-level routine is used by many algorithms of PT-Scotch, for instance to share matching data in the coarse graph building routine (see Section 3.1).

Because global and local indexings coexist, two adjacency arrays are in fact maintained on every process. The first one, usually provided by the user, holds the global indices of the neighbors of any given vertex, while the second one, which is internally maintained by PT-Scotch, holds the local and ghost indices of the neighbors. Since only local vertices are processed by the distributed algorithms, the adjacency of ghost vertices is never stored on the processes, which guarantees the scalability of the data structure as no process will store information of a size larger than its number of local outgoing arcs.

2.6.3 Experimental conditions

Because the works reported here span on a duration of almost ten years, our experiments were performed on various hardware platforms, ranging from a laptop (for sequential measurements) to modern large-scale parallel systems, on up to 2048 processing elements in some cases. Run times vary accordingly, as well as with respect to the successive versions of Scotch, which has undergone many changes and improvements along with time.

The test graphs which we used during our experiments are listed in Table 2.1. Most of them are freely available, on the Internet, to any people willing to compare its results with ours. Many of them can be found in the University of Florida matrix collection [28]. Some others were gathered during the course of the former European Project Parasol [91].

[13] Yet, some authors [121] report to have used more complex procedures, involving hash tables and binary search, for global-to-local index conversion.

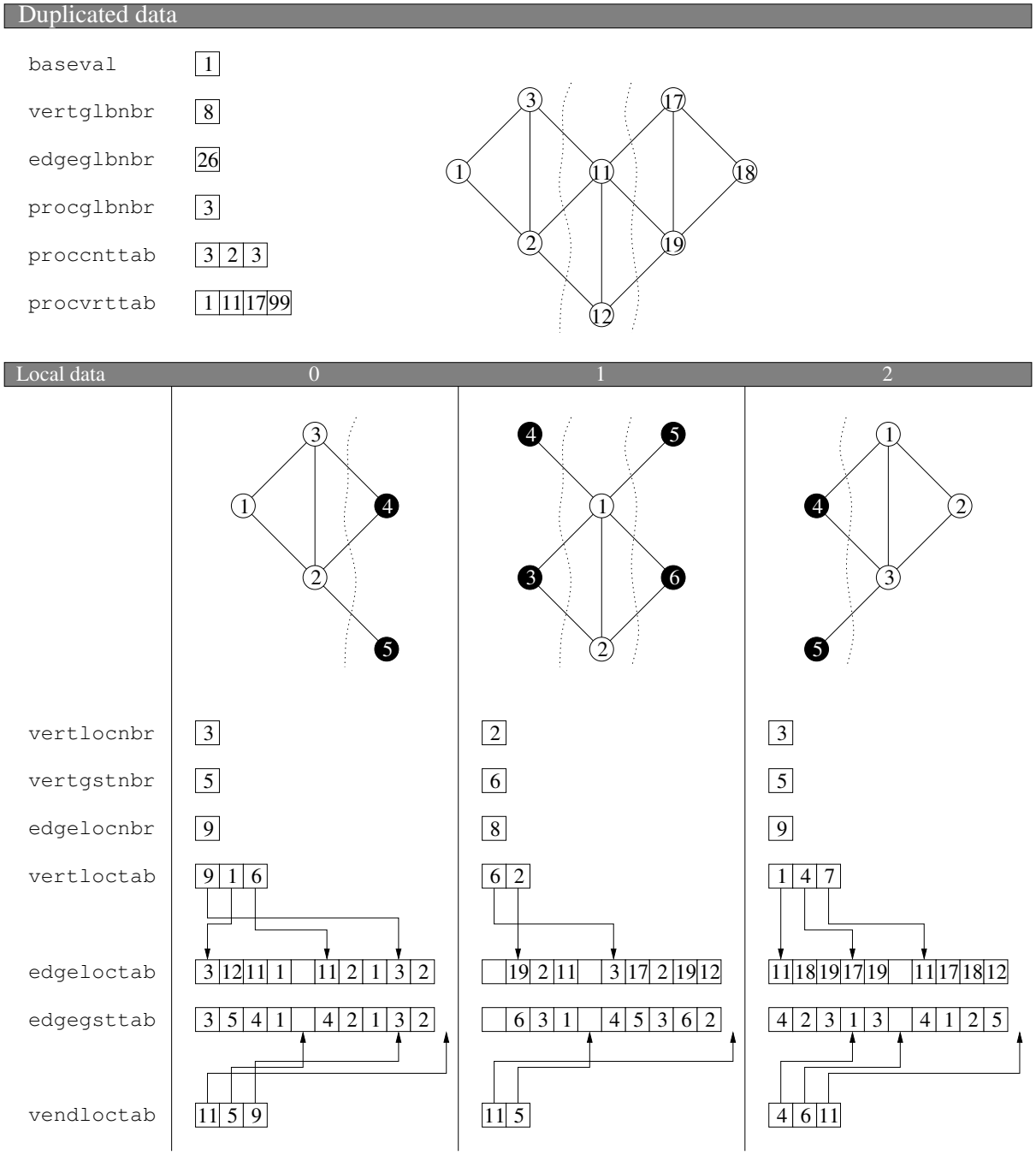


Figure 2.10: Data structures of a graph distributed across three processes. The global image of the graph is shown above, while the three partial subgraphs owned by the three processes are represented below. Adjacency arrays with global vertex indexes are stored in `edgeloctab` arrays, while local compact numberings of local and ghost neighbor vertices are internally available in `edgegsttab` arrays. Local vertices owned by every process are drawn in white, while ghost vertices are drawn in black. For each local vertex i located on process p, the global index of which is (`procvrttab`$[p]+i-$`baseval`), the starting index of the adjacency array of i in `edgeloctab` (global indices) or `edgegsttab` (local indices) is given by `vertloctab`$[i]$, and its after-end index by `vendloctab`$[i]$. For instance, local vertex 2 on process 1 is global vertex 12; its start index in the adjacency arrays is 2 and its after-end index is 5; it has therefore 3 neighbors, the global indices of which are 19, 2 and 11 in `edgeloctab`.

Graph	Size($\times 10^3$)		Avg. degree	Graph	Size($\times 10^3$)		Avg. degree
	V	E			V	E	
10millions	10424	78649	15.09	coupole5000	1105	26036	47.12
144	145	1074	14.86	coupole8000	1768	41657	47.12
23millions	23114	175686	15.20	crankseg1	53	5281	200.01
3elt	5	14	5.81	crankseg2	64	7043	220.64
45millions	45241	335749	14.84	gupta1	32	1066	67.05
4elt	16	46	5.88	inline1	504	18156	72.09
4elt2	11	33	5.89	invextrusion1	30	907	59.64
598a	111	742	13.37	lhr71c	70	1527	43.43
82millions	82294	609508	14.81	m14b	215	1679	15.64
aatken	43	88	4.14	mixingtank	30	983	65.60
altr4	26	163	12.50	mt1	98	4828	98.96
audikw1	944	38354	81.28	ocean	143	410	5.71
auto	449	3315	14.77	oilpan	74	1762	47.77
b5tuer	163	3874	47.64	pwt	37	145	7.93
bbmat	39	1274	65.77	qimonda07	8613	29143	6.76
bcsstk29	14	303	43.27	quer	59	1404	47.48
bcsstk30	29	1007	69.65	rotor	100	662	13.30
bcsstk31	36	573	32.20	s3dkq4m2	90	2365	52.30
bcsstk32	45	985	44.16	s3rmt3m3	5	101	37.77
bmw32	227	5531	48.65	ship001	35	2305	132.00
bmw7st1	141	3599	50.93	ship003	122	3982	65.43
bmwcra1	149	5248	70.55	shipsec1	141	3836	54.46
body	45	164	7.26	shipsec5	180	4967	55.23
bracket	63	367	11.71	shipsec8	115	3269	56.90
brgm	3699	151940	82.14	sphere	16	49	6.00
bump	10	29	5.92	thread	30	2220	149.32
cage15	5154	47022	18.24	tooth	78	453	11.58
chanel1m	81	527	13.07	wang3	26	76	5.80
conesphere1m	1055	8023	15.21	x104	108	5030	92.81

Table 2.1: Test graphs which we used in all of the experiments reported in this dissertation.

Because many experiments in science are now performed by means of software, it is of utmost importance that software be released along with the publications which rely on it. Otherwise, peer referring is no longer possible and, since software can take time to be recoded, scientific progress is slowed down. For all these reasons, since its version 4.0, our Scotch software has been released as free/libre software. Version 5 of Scotch is distributed under the CeCILL-C license [16], which has basically the same features as the GNU LGPL (*Lesser General Public License*) [45]: ability to link the code as a library to any free/libre or even proprietary software, ability to modify the code and to redistribute these modifications.

Also, Scotch has been designed in a highly modular way, so that new methods can be easily added to it, in order for it to be used as a testbed for the design of new partitioning and ordering methods [97].

For the sake of reproducibility and ease of comparison, the random generator used in Scotch is initialized with a fixed seed. This feature is essential to end users, who can more easily

reproduce their experiments and debug their own software, and is not significant in term of performance. Indeed, in the case of sparse matrix ordering, we have experimented that the maximum variation of ordering quality, in term of OPC, between 10 runs performed with varying random seed, was less than 2.2 percent on our test graphs [24]. Consequently, we did not find necessary to perform multiple runs of SCOTCH to average quality measures.

Chapter 3

Graph coarsening algorithms

As said in the previous chapter, multilevel schemes are extremely powerful tools for graph partitioning, provided that unbiased coarsening and local optimization algorithms are defined for the targeted classes of graphs.

In the next section, we will describe the ways in which the parallel graph partitioning problem has been addressed in the literature, with respect to each of its key issues: matching, folding, and centralization of the distributed data for the computation of the initial partition on the coarsest graph. Then, in the following sections, we will present our contributions to these issues.

3.1 State of the art

The process of creating a coarser distributed graph from some finer distributed graph is quite straightforward, once every fine vertex has been assigned to the cluster representing its future coarse vertex. By performing a halo exchange of the cluster indices borne by fine vertices, each of the latter can know the indices of the clusters to which its neighbors belong. The coarse adjacencies of each of the fine vertices contributing to some cluster can then be gathered on the respective processes which will host each of the resulting coarse vertices, such that coarse adjacencies of coarse vertices can be determined by removing duplicate coarse neighbors from the received neighbor lists.

Several methods can be used to create the vertex clusters. All of them use graph adjacency to mate vertices which are as close to each other as possible, so as to preserve the high-level topological structure of the finer graphs. Otherwise, the prolongation to the finer graphs of partitions computed on the coarser graphs would not be likely to represent good solutions of the original problem, because of the biases introduced by coarsening artifacts. While large clusters may result in less coarsening levels, therefore potentially reducing the accumulation of artifacts over the levels, they can also be a source of artifacts if they are not compact enough. Because the creation of large compact clusters in parallel is more complex and more likely to require more communication than for smaller clusters, most parallel coarsening methods are based on edge matchings, which result in clusters comprising at most two vertices.

The key issue for parallel graph coarsening is therefore to devise parallel matching algorithms which are both efficient and unbiased, that is, which provide coarse graphs the topological properties of which are independent of the distribution of fine vertices across the processes of the parallel program.

3.1.1 Parallel matching algorithms

Parallel matching algorithms are a very active research topic [17]. In our context, this problem is relaxed, because matchings do not need to be maximal. Instead, they should exhibit randomness properties suitable to the preservation, by the coarsened graphs, of the topological properties of the finer graphs to which they are applied.

The critical issue for the computation of matchings on distributed graphs is the breaking of mating decision ties for edges spanning across processes. Any process willing to mate one of its local vertices to a remote adjacent vertex has to perform some form of query-reply communication with the process holding the remote vertex. Because of communication latency and overhead, mating requests are usually aggregated per neighbor process, resulting in a two-phase algorithm. In the first phase, processes send mating requests to their neighbors. In the second phase, neighbors answer positively or negatively depending whether the requested vertices are free, have already been matched in a previous phase, or are temporarily unavailable because they have themselves requested another remote vertex for mating. The asynchronicity between requests and replies creates ties and dependency chains which hinder the convergence of the algorithm. For instance, when several vertices located on different processes request the same remote vertex, at most only one of the requests can be satisfied. Moreover, when the requested vertex has itself asked for another vertex, none of the incoming requests will be satisfied, because the solicited vertex cannot know whether its own request will be satisfied or not before replying to its applicants. This phenomenon increases along with the probability of neighbor vertices to be remote: when graphs are arbitrarily distributed (which is usually the case before they are partitioned for the first time), when they have high degrees, and/or when the number of processes is high.

Several solutions to the tie-breaking problem have been proposed in the literature, which differ in the granularity of the remote dependency exclusion mechanism.

Tie breaking at the vertex level

The first solution for tie-breaking at the vertex level was proposed by Barnard [7] in the context of a parallel formulation of a multilevel spectral partitioning algorithm [103], and based on a parallel formulation of Luby's graph coloring algorithm [82] to compute a maximal independent set of vertices. The rationale behind the use of a graph coloring is to completely avoid dependency chains, by preventing potential sought-after vertices from being applicants themselves.

In the matching algorithm, each of the colors is considered in turn, and only vertices of the current color can perform mating requests. Since, by definition, independent set coloring guarantees that no two neighbor vertices have the same color, potential mates cannot be applicants, and thus will always be able to answer positively to one of their own applicants. Colors are processed in a round-robin way until all vertices are matched or have all of their neighbors matched, or until some minimum coarsening threshold is reached [66, 71]. This algorithm does not guarantee that collisions between requests will not happen, since two vertices of the same color can still send a request to a same common neighbor vertex of another color; however, reducing the number of vertices able to send a request during each time step significantly reduces the number of such collisions.

The distributed formulation of Luby's algorithm works as follows. First, every graph vertex is assigned a random number, and a halo exchange is performed so that all vertices know the number borne by all of their neighbors, whether local or remote. Then, vertices having the highest number among all of their neighbors are painted with the first color, after which these

998699	908307	848574	804471	777182	757239	743009	733290
720049	700112	661773	590857	478982	338154	200174	99186
41975	15071	4750	1377	372	108	20	6

Table 3.1: Sizes of Luby's color sets for graph 10MILLIONS. Our test implementation of Luby's algorithm, applied to a randomly permuted vertex set, yields 24 color sets with the above distribution. Data extracted from [58].

vertices are removed from the graph. Remaining vertices having the highest number among all of their remaining neighbors are painted with the second color and removed, and so on until all of the vertices have been considered.

This algorithm is simple and elegant, but it has some drawbacks. The sizes of the color sets are usually unbalanced, the last ones being much smaller than the first ones; it is often the case that the last set comprises only one single vertex.

This imbalanced distribution adversely affects convergence time, both for computing the coloring and for performing the matching. Because the last rounds of the coloring algorithm involve small sets of vertices, little useful work is carried out in comparison to the induced communication overhead which is itself dominated by network latency, resulting in poor scalability. This is why, in at least one implementation [71], the coloring algorithm is halted prematurely, the loss in coarsening quality resulting from the absence in the mating process of the remaining small fraction of the vertices having been judged negligible with respect to the overall gain in run time. Regarding the matching algorithm, larger sets increase the probability that multiple requests are directed towards the same vertices, therefore reducing their probability of success, while smaller sets again result in excessive communication overhead. A solution to this problem could be to rework Luby's algorithm so as to improve the balance of color sets. However, this is not likely to reduce much the number of color sets, that is, the number of communication rounds.

All these reasons have motivated the early replacement in PARMETIS of the aforementioned implementation of Luby's algorithm by a more straightforward algorithm [66, Section 4], almost similar in outcome to the probabilistic matching algorithm that we devised and which will be described in Section 3.4. Karypis' modified algorithm works as follows. In the beginning, all local and halo vertices of every process are flagged as unmatched. The algorithm performs a fixed number of passes (4, in the current implementation), during which all remaining unmatched local vertices are considered in randomized order, by means of a precomputed random permutation. Every considered vertex selects a potential mate among its yet unmatched neighbors linked by edges of heaviest weight, according to the heavy edge matching policy [71]. If the mate is local, both vertices are immediately flagged as matched with each other. Otherwise, the behavior of the algorithm depends whether some condition, based on the pass number and on the respective global indices of the requesting vertex and of its potential mate, is satisfied. If it is, a mating request is stored into the proper communication buffer, to be sent to the relevant process at the end of the pass, and the local vertex is flagged as busy. Otherwise, nothing happens. However, in both cases, the halo vertex representing the remote mate is flagged as busy, so that no other local vertices can ask it for mating. At the end of the pass, mating request messages are exchanged and processed in random order. Mating requests are accepted or rejected by the destination processes, depending whether their local vertices have already been matched or not, and corresponding replies are aggregated and sent back.

If the aforementioned condition were not present, all possible remote matings would lead to the sending of mating requests, thus saturating the system as all senders would not be willing to mate as receivers. By preventing roughly half of the requests from being sent, and by letting

their senders remain free to mate as receivers, the algorithm increases convergence speed. This will be discussed further in Section 3.4.

Tie breaking at the subdomain level

This section discusses attempts to avoid as much communication as possible during the matching process, so as to reduce the impact of dependency chains on the convergence of the matching algorithm. This result is sought for by privileging local matings over remote ones, that is, by restricting matings of vertices to their local subdomains. In this case, like in the fully sequential case, tie breaking is naturally resolved by the sequentiality of computations within each subdomain.

This approach has been followed by Walshaw *et al.* in [121]. However, the efficiency of this method heavily depends on the initial distribution of the vertices across the processes of the parallel partitioner. In the case of random distribution, the probability of two neighbors being located on the same process decreases along with the number of processes. This is why, as a fall-back strategy, Walshaw *et al.* also provide an iterative remote mating algorithm to post-process remaining unmatched vertices [122].

Indeed, depending on the distribution of graph vertices, local matching can induce heavy topological bias in the coarsened graphs. Consequently, its use must be restricted to cases where this bias is likely to be minimal, that is, when the number of ghost vertices (the possible remote mates) is small enough compared to to the number of local vertices. This may be true only in the case of dynamic repartitioning of graphs which have already been partitioned, or for graphs which have been distributed by means of low-cost, geometrical partitioning, when geometrical data are available and when the mesh geometry is simple enough to be efficiently cut by planes. In the case of repartitioning, local matching may also be necessary to be sure that all fine vertices of some coarse vertex belong to the same subdomain, in order to compute accurately migration costs of the coarse vertices [107].

3.2 Folding

In the course of the coarsening phase of a multilevel framework, folding refers to the process of redistributing the vertices of a coarse graph onto a smaller number of processes than those holding the finer graph. Performing such a folding has several benefits. First, it can reduce communication overhead. As coarsening goes on, the number of vertices per process is bound to decrease, and communication time tends to be dominated by start-up time and latency. Using too many processes to handle so little data makes collective communication prone to operating system hazards, and increases the probability of increased latency. Second, concentrating vertices on fewer processes is most likely to decrease the number of remote neighbors per process, resulting in shorter messages. Third, for algorithms biased towards local computations (see the above section), increasing the number of local vertices can improve the quality of the output they produce [70].

In all of the multilevel implementations that we are aware of, when a folded subgraph is created on a subset of the processes, the rest of the processes remain idle until they are required again during the uncoarsening phase [66]. However, they can be used in parallel to compute the initial partition at the coarsest level, when coarse distributed graphs are small enough to be able to be centralized on a single processor (see Section 3.5).

3.3 Centralization

Once the distributed graph is coarsened down to a prescribed, small size, an initial partition has to be computed. While fully parallel methods for an initial partitioning could be used, such as parallel genetic algorithms (see Section 4.1.2), all implementations we are aware of resort to state-of-the-art sequential methods, because of the very high quality of the partitions they produce, at reasonable cost for such small graphs. Since the initial partition will only be subject to refinement and will never be reconsidered globally, quality concerns dominate at this stage.

When folding is used, the coarsest graphs are already concentrated on a small number of processes, and centralizing them on a single process is not much work. Yet, all the remaining processes are idle, and may be used to speed-up the computation of the initial partition. In [71], the centralized graph is duplicated on all processes, and each of them computes sequentially the branch of the recursive bipartitioning tree leading to the subdomain it is in charge of. Let $T(x)$ be the cost of computing a bipartition on some fraction x of the centralized graph, linear in x. With the above approach, instead of computing the initial partition in $T(1) + 2T\left(\frac{1}{2}\right) + 4T\left(\frac{1}{4}\right) + \cdots$, which leads to a computation time in $\log(p)T(1)$, each branch is independently computed in $T(1) + T\left(\frac{1}{2}\right) + T\left(\frac{1}{4}\right) + \cdots = 2T(1)$.

The gain, albeit real in percentage, is small in absolute value, as it only concerns small graphs. Moreover, this approach can only be used when successive bipartitioning processes are independent, which prevents it from being used in the context of static mapping [93, 96].

3.4 Parallel probabilistic matching

The approach we have chosen to the parallel matching problem follows a Monte-Carlo approach. This idea, first exposed by Chevalier in [22], consists in allowing a vertex to send a request only when the random number it draws is above some threshold. In [22] and [24], this threshold depended on the numbers of local and remote neighbors of the vertex. In [58], we opted for a more straightforward algorithm, which does not take into consideration any parameter linked to the graph distribution, and is consequently less susceptible to distribution biases: every unmatched vertex can send a mating request with probability 0.5, or remain silent.

Our algorithm, which consists of successive matching rounds, works as follows. Initially, all local vertices are put in a wait queue, in random order. During a matching round, all queued vertices are processed one by one. For each vertex, a random bit value is considered. If it is zero, the vertex is put back into the queue, else one of its yet unmatched neighbors of highest edge weight is randomly selected for mating, according to the heavy edge matching policy. If the selected vertex is local, the matching is immediately accepted; else, a mating request is enqueued in the message destined to its owner process. After all vertices have been considered, mating request messages are exchanged between processes, and are processed in random order by their recipients. Each of the requests is considered in message order. If the sought-after vertex is itself a sender, no reply will be returned, indicating collision; the matching did not succeed this round, but may well succeed next time. Otherwise, if the vertex is not already matched, the matching is accepted; else, a negative answer is crafted so that the halo copy of the remote vertex will be flagged as already matched by the sending process (however, other processes may not yet be aware of this information until one of their local vertices sends a request directed towards this vertex). Then, reply messages are sent back, and mating data for local and ghost vertices are updated accordingly. The above communication round is repeated several times, after which a final local matching sweep is performed to match locally, either with a local unmatched neighbor or with itself, every vertex remaining in the queue.

Albeit motivated by the same rationale (reducing communication overhead and leaving some vertices idle to increase the probability of successful mating during each pass), our algorithm differs in some aspects from the non color-based matching algorithm of ParMeTiS (see page 25).

First, the decision of keeping a vertex active or not is taken individually per vertex, on a random basis, and does not depend on its relations with some remote potential mate.

Second, this activity test concerns all vertices, also including internal vertices, that is, those which have no remote neighbor. While this may seem useless at first sight, because matings between these vertices and their neighbors would be immediately accepted, thus speeding up convergence, it prevents a subtle bias that would impact border vertices. Indeed, in the case of ParMeTiS, border vertices are active only half of the time when willing to mate with remote neighbors, while their local matings are always immediately accepted, as are those of internal vertices. Half the time, a border vertex willing to mate with a remote vertex will remain inactive and flagged as unmatched, leaving it free to be matched locally by subsequent local vertices before any remote mating message arrives. Moreover, its ghost mate will be flagged as busy, preventing neighboring border vertices to mate it, and therefore also favoring their local matching. All of the internal vertices will be matched at the end of the first pass, so that, starting from the second pass, only border vertices will be processed, for which local matchings with neighboring border vertices will again be privileged.

In order to evaluate the convergence speed of our algorithm, we have instrumented our matching algorithm to output two values after each round. The first value is the ratio of matched vertices, expressed as a percentage of the total number of fine vertices. This value is obviously equal to 100 % after the final pass. The second value is the coarsening ratio among matched vertices, that is, the number of coarse vertices computed to date, divided by the number of processed fine vertices. By nature, this value ranges between 50 %, in the ideal case where all fine vertices have been paired into coarse vertices, and 100 %, in the case where no neighbors could be found and all coarse vertices each comprise one single fine vertex.

Table 3.2 presents the data collected when recursively coarsening our test graphs down to one thousand of vertices per process, for a number of processes ranging from 2 to 512. It shows the mean value and the mean absolute deviation of each of the two aforementioned values, for each of the collective rounds and for the final local round[1]. The number of collective rounds has been set to 5, to keep it small comparatively to the size of the color sets computed by Luby's algorithm; it is just one more than the number of rounds in the matching algorithm of ParMeTiS. Assuming, as a rule of thumb, a matching probability a bit lower than 0.5 (depending on the probability of the requested vertex to be inactive and of the probability of collision between matching requests, based on graph topology), 5 collective passes were supposed to be enough to match more than 80 % of the vertices.

Experimental figures corroborate our assumptions and validate our approach. Five collective rounds are enough to match more than 80 % of the graph vertices, with a low resulting coarsening ratio of 53.7 %. This ratio indicates that remote mating is efficient, and that no topological biases, due to initial graph data distribution across processes, are likely to occur. Consequently, the final, local round is not likely to induce an important topological bias, since it only involves 15 % of the vertices on average, after many remote matings have been performed. A sixth collective round has been experimented with, but did not significantly change partition results.

This algorithm has been introduced in PT-Scotch quite recently, as it is only available since revision 5.1.6. It provides a significant improvement in speed compared to the previous

[1]Using such a local round will result in the same biases as those criticized in the implementation of the matching algorithm of ParMeTiS. Yet, unlike for the latter, these biases are not inherent to the algorithm, but depend on the number of collective rounds that the user is willing to perform.

Pass	Matching		Coarsening	
	Avg.	M.a.d.	Avg.	M.a.d.
C1	53.3	12.3	50.4	0.7
C2	68.7	13.6	51.6	2.2
C3	76.2	12.2	52.5	3.3
C4	81.0	10.6	53.2	4.0
C5	84.5	9.1	53.7	4.5
LF	100.0	0.0	59.4	6.8

Table 3.2: Average and mean absolute deviation of the percentage of the vertices processed, and of the coarsening ratio of the processed vertices, after each of five collective matching rounds (C) and after the local final (LF) round. These data were collected by recursively coarsening our test graphs on numbers of processes ranging from 2 to 512. Data extracted from [58].

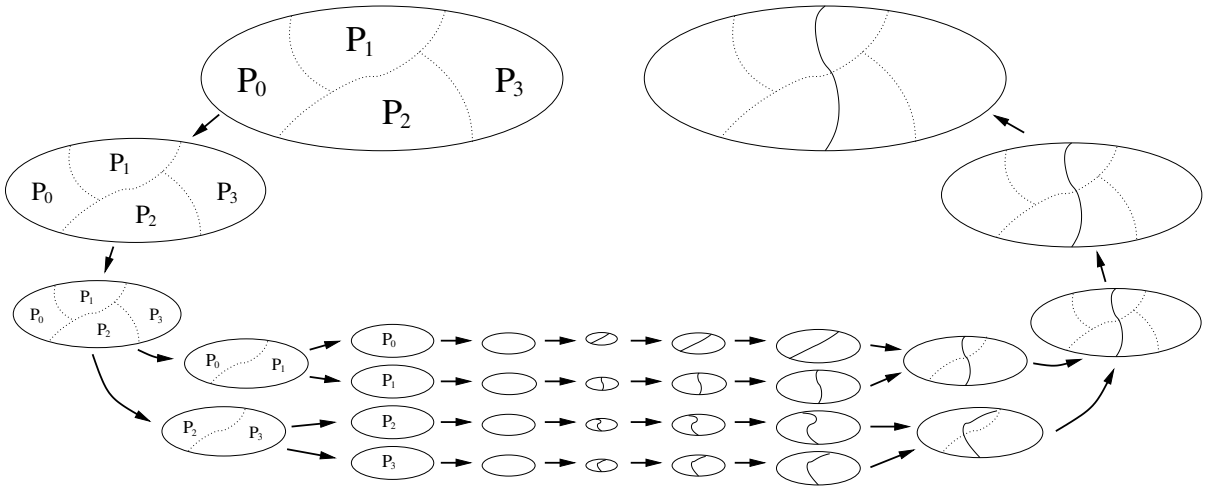


Figure 3.1: Diagram of the parallel computation of the separator of a graph distributed across four processes, by parallel coarsening with folding-with-duplication, multi-sequential computation of initial partitions that are locally prolonged back and refined on every process, and then parallel uncoarsening of the best partition encountered.

version.

3.5 Folding with duplication

One of the original features of our multilevel framework, as opposed to competing implementations (see Section 3.2), is its ability to duplicate coarse graphs during the folding process, so that no process remains idle. The coarsening phase starts once the matching phase has completed. It can be parametrized so as to allow users to choose between two options. In the first case, all coarsened vertices are kept on their local processes (that is, processes that hold at least one of the ends of the coarsened edges), as shown in the first steps of Figure 3.1, which decreases the number of vertices owned by every process and speeds-up future computations. In the second case, coarsened graphs are folded and duplicated, as shown in the next steps of Figure 3.1, which increases the number of working copies of the graph and can thus reduce communication and improve the final quality of the separators.

In fact, separator computation algorithms, which are local heuristics, heavily depend on the quality of the coarsened graphs, and we have observed with the sequential version of Scotch that taking every time the best partition among two, obtained from two fully independent multilevel runs, usually improves overall ordering quality. By enabling the folding-with-duplication routine (which will be referred to as "fold-dup" in the following) in the first coarsening levels, one can implement this approach in parallel, every subgroup of processes that hold a working copy of the graph being able to perform an almost-complete independent multilevel computation, save for the very first level which is shared by all subgroups, for the second which is shared by half of the subgroups, and so on.

The problem with the fold-dup approach is that it consumes a lot of memory. When no folding occurs, and in the ideal case of a perfect and evenly balanced matching, the coarsening process yields on every process a part of the coarser graph which is half the size of the finer graph, and so on, such that the overall memory footprint on every process is about twice the size of the original graph. When folding occurs, every process receives two coarsened parts, one of which belongs to another process, such that the size of the folded part is about that of the finer graph. The footprint of the fold-dup scheme is therefore logarithmic in the number of processes[2], and may consume all available memory as this number increases. Consequently, as in [66], a good strategy can be to resort to folding only when the number of vertices of the graph to be considered reaches some minimum threshold. This threshold allows one to set up a trade-off between the level of completeness of the independent multilevel runs which result from the early stages of the fold-dup process, which impact partitioning quality, and the amount of memory to be used in the process.

Our fold-dup implementation is also insensitive to the number of processes on which it is run. In the case of an uneven number of processes, one of the process subsets has simply one more element than the other. The smaller subset on which the coarse graph is folded is of course likely to take more time to perform the same amount of computation, but this is a negligible percentage for large a number of processes, and a small amount of computation for a small number of processes. Anyway, the cost is not greater than that of running the program on a number of processes equal to the immediately smaller power of two, and this penalty only happens for the stages of coarsening where folding occurs, that is, where the amount of computation is small.

3.6 Multi-centralization

After the final stage of our fold-dup method, no more than two processes may have the same coarse graph, since the two copies of a graph folded at some stage undergo afterward an independent coarsening process, using a different pseudo-random seed. Unlike in [71], where all centralized graphs are identical and the same bipartitions must be computed on all processes, we favor the computation of different initial partitions by each of the processes.

This multi-sequential phase is illustrated at the bottom of Figure 3.1: the routines of the sequential Scotch library are used on every process to complete the coarsening process, compute an initial partition, and prolong it back up to the largest centralized coarsened graph stored on each of the processes. Then, the partitions are prolonged back in parallel to the finer distributed graphs, selecting the best partition between the two available when prolonging to a level where fold-dup had been performed. This distributed prolongation process is repeated until we obtain a partition of the original graph.

[2]The computations leading to this result are equivalent to those in page 27.

The benefits of producing different initial partitions have been shown in [22]. Further results integrating the use of fold-dup will be presented in the next chapter.

Chapter 4

Partition refinement algorithms

As discussed above, multilevel schemes can be extremely powerful tools for graph partitioning, provided that unbiased coarsening and partition refinement algorithms are defined for the targeted classes of graphs. While several authors had already proposed scalable solutions for the parallelization of the coarsening phase, as seen in the previous chapter, the situation was not as satisfactory for refinement algorithms, so that several of my contributions have specifically addressed this topic.

4.1 State of the art

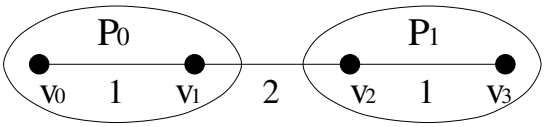
Because the state-of-the-art local optimization methods used for partition refinement in sequential multilevel frameworks were so robust and cost-effective, much effort has been put on their parallelization. Yet, it is common knowledge in the parallel computing community that the best parallel algorithm to solve a given problem is most often not the parallel transposition of the best sequential algorithm for this purpose. Partition refinement by no means escapes this rule, as state-of-the-art sequential methods bear strong sequentiality constraints which prevent their direct transposition into scalable parallel formulations, so that global methods have also been considered.

4.1.1 Parallelization of local partition refinement methods

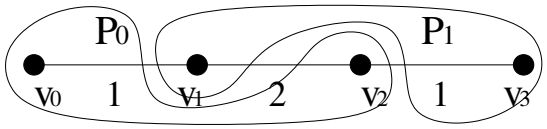
The issue of colliding moves

As seen in Section 2.5.1, local optimization algorithms for partition refinement are based on successive moves of frontier vertices to decrease the current value of the prescribed cost function. In the sequential case, vertices are moved one by one, and the gains of the selected vertex and of its neighbors are recomputed on the fly, such that one always knows exactly the outcome of a sequence of moves. However, this no longer holds if moves are to be performed independently in parallel, as illustrated by the infamous "neighbors swap" case of Figure 4.1.

If the two vertices had been far away one from one another, the problem described above would not have shown up. What causes it is that the two vertices are neighbors, and that the mutual impact of their moves cannot be accounted for to avoid performing the second move as soon as the first one takes place. Indeed, if the two vertices are distributed across separate processors, there is no immediate means for each of the processes running the algorithm to inform its peers; this is not desirable either, as taking into account the constraint of having to update the gain values of neighbor vertices would create time dependencies which would dramatically hinder the scalability of the algorithm.



(a) Current bipartition of a weighted graph, with $f_C = 2$, $\nabla f_C(v_0) = +1$, $\nabla f_C(v_1) = -1$, $\nabla f_C(v_2) = -1$, $\nabla f_C(v_3) = +1$.



(b) Resulting partition after moving simultaneously v_1 and v_2. While each of them had negative gain, $f_C = 4$.

Figure 4.1: Adverse impact on the cost function of the simultaneous move of two vertices which, moved individually, would have improved it.

It follows from the previous example that, for any parallel version of some FM-like local optimization algorithm to work, no two neighbor vertices must ever be considered at the same time by distinct processing elements. This is indeed a very strong constraint, which imposes a heavy burden on the programmer. Since dynamic remote locking mechanisms would generate too much communication, this constraint must be enforced a priori, with certainty, within the core of the algorithm itself. Several solutions have been proposed in the literature, which differ by the granularity of the exclusion mechanism, in a way very similar to the matching problem (see Section 3.1.1). All of them are based on rounds within which the exclusion constraint will be enforced, with some synchronization taking place at the end of the rounds so as to recompute the gains of remote neighbors.

Collision exclusion at the vertex level

The first solution, proposed by Karypis and Kumar, enforces neighbor exclusion at the vertex level [71]. It is based on a coloring of the distributed graph, computed by means of a distributed variant of Luby's algorithm (see Section 3.1.1), to split vertices into groups such that no two neighbor vertices belong to the same group. The groups associated with the colors are then processed in a round-robin fashion, and within each round all vertices of negative gain are moved to their preferred destination part. Then, vertex movement information is exchanged between neighboring processes, so that the gains of their neighbors can be recomputed before the next round begins. Vertices are not physically moved from one process to another when changing their part, but have their subdomain tag updated on their owner process, as well as on all of the neighbor processes which maintain a halo copy of the vertex as a remote neighbor of their local vertices.

While this method is very scalable, as computations are most likely to be evenly spread across processes (assuming that the number of vertices of the same color is approximately the same on every process), it loses the hill-climbing capabilities of the original sequential algorithm, resulting in poorer partition quality. In the sequential version, the algorithm could accept moves of smallest positive gains, with the expectation that further moves of neighboring vertices would decrease the cost function to a lower value. By uncoupling moves of neighboring vertices into different rounds which may not be performed just one after the other, the parallel algorithm forgoes the possibility to accept moves of positive gain. Otherwise, many processes could decide simultaneously, within the same round, to select vertices of positive gain, which might not be of overall minimum positive gain, without any chance for the algorithm to compensate later for these moves. Consequently, this parallel version can only behave as a gradient-like algorithm.

Indeed, as the number of communication rounds induced by the vertex coloring can be very large, as already evidenced in Section 3.1.1 in the case of matching, this color-based, vertex-level

exclusion process has also been abandoned in ParMeTiS, in favor of a variant of the second solution described below.

Collision exclusion at the subdomain level

The second solution, proposed by Walshaw and Cross, enforces neighbor exclusion at the subdomain interface level [119]. Here again, the algorithm takes the form of two nested loops. At the outer level, interface regions between different subdomains are separated into disjoint areas, which are centralized on one of their two owner processes. These subgraphs are treated as independent problems, onto which sequential optimization algorithms are applied. All of the moves performed locally on the subgraphs are then prolonged back to the full distributed graph, and this process goes on until convergence is achieved.

This algorithm preserves, in each interface subgraph, the hill-climbing capabilities of the sequential local optimization algorithm, in this case a KL-like algorithm. Although it has only a limited effect because the interface regions are generally long and thin, it can help smooth the frontiers, by allowing vertices to line up, along the mesh structure, with their already moved neighbors.

However, what is gained in quality is lost in runtime efficiency. Interface subgraphs are assigned to processes in an arbitrary way, which may result in heavy load imbalance, all the more that some faces may require more work than others. Moreover, with this method, vertices are physically moved when changing their part, such that each process owns only the vertices of the subdomain it represents. This can cause additional load imbalance, as subdomains having fewer but heavier vertices represent less work in term of graph algorithms than subdomains filled with many light vertices, but also induces more communication. All of these factors tend to result in poor overall scalability [119, page 1653]. Furthermore, this one-to-one mapping between subdomains and processes hinders the ability of the algorithm to run efficiently on a number of processors which is not a divisor of the number of subdomains. This design choice comes from the fact that Jostle was mainly devised as a parallel dynamic repartitioner.

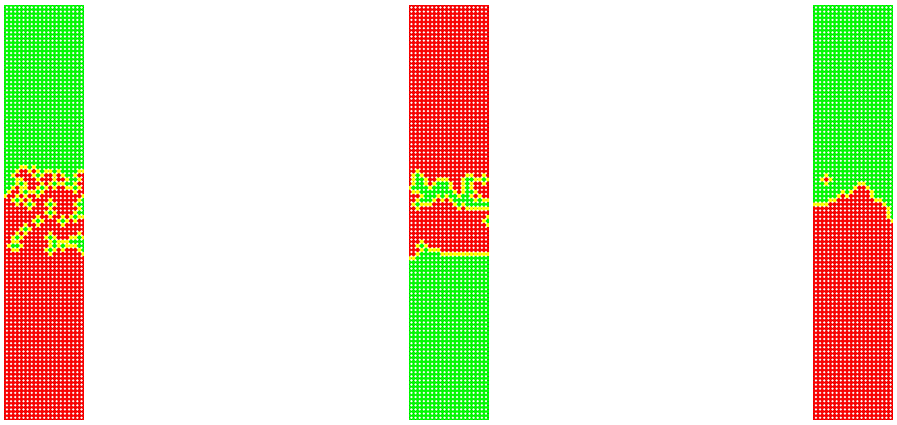
In the method actually implemented in ParMeTiS [66, Section 4], interfaces are not considered explicitly. Instead, for each pass, all vertex moves which would improve the cut are performed provided that some condition, depending on the pass number and on a randomized function of the numbers of the origin and destination subdomains, is satisfied. The purpose of this condition is to exclude collisions at the subdomain level: during some pass, for all origin and destination subdomains, moves are only accepted in one of the two possible directions, which changes along with the parity of the pass number. Consequently, neighbor swaps cannot happen, because one of the two moves will always be rejected. While this solution is much easier to implement than interface centralization, it can only perform gradient-like improvements, without any hill-climbing capabilities.

4.1.2 Parallel global partition refinement methods

Since the road to parallel local optimization methods was so full of hurdles, several authors have investigated the path of global methods. These methods, which are too expensive to be applied to large graphs in the sequential case (see Section 2.5.3 for an overview of the most popular ones), most often have a strong potential for parallelization. Yet, due to their global nature, their cost may be high even in parallel, and their use within a multilevel framework somehow goes against the nature of this framework, which is designed to use fast algorithms that operate on reduced problem spaces.

Refinement by genetic algorithms

The class of global methods which has been used the most extensively in the literature is evolutionary algorithms, and more specifically genetic algorithms [4, 13, 104, 105, 113]. Genetic algorithms (GA) are meta-heuristics used to solve multi-criteria optimization problems using an evolutionary method (see [125] for a good introduction to these methods). They consist in simulating iteratively, generation after generation, the evolution of a population of individuals which represent solutions to the given problem, selecting best-fitting individuals as candidates for breeding the next generation. GA are known to converge very slowly and cannot therefore be applied to large graphs [4, 13], as evidenced in Figure 4.2, which illustrates both the versatility of GA, that is, their ability to converge even with very naive criteria, and the very slow rate of this convergence in the absence of specialized local optimization methods.



(a) Best individual of the first generation. The separator is long and irregular.

(b) Best individual of the hundredth generation. The size of the separator decreases, and its contour gets smoother.

(c) Best individual of the thousandth generation. Parts are now almost connected and separator size is closer to the optimal.

Figure 4.2: Example of results which can be obtained when using a bare GA, without any multilevel or local optimization heuristics, to search for a vertex separator on a simple 2D grid. Figure extracted from [22].

Talbi and Bessière [113] were among the first to use parallel GA to solve the graph partitioning problem. In their implementation, the individuals which represent solutions to the k-way graph partitioning problem are encoded in the form of a linear array of size equal to the number of graph vertices, each cell of which holds the number of the part to which the associated vertex is assigned. The population is evenly spread across all processes. In order to reduce communication and increase concurrency, all of the individuals which are located on the same process are considered as an isolated population (also called *deme*) living on an island [126]. Only occasionally can a few "champions" move from one island to another, to propagate their successful chromosomes into other populations which could have been trapped in local optima. The exchange of champions is performed between neighboring processors only, across the links of the torus-shaped communication system of their Supernode Transputer-based machine. This limitation in communication allows them to achieve linear speed-up up to the 64 processors of their machine.

In the context of mesh partitioning for FEM computations, for which vertex coordinates are available, several authors have used different, more compact representations of individuals to

model geometric bipartitions. For instance, Khan and Topping consider a population of cutting planes to bisect recursively the problem space [74], while Rama Mohan Rao [104] represents a bipartition as a force field created by two point-charges (defined by only 8 numbers: two coordinate sets for the points and two scalars for intensities), used as a separator. The GA computations of this latter implementation are performed in parallel according to a multi-deme distribution of the population. However, unlike for Talbi and Bessière's implementation, champions are exchanged between all processes at the end of local computations.

In order to speed-up the convergence of GA, the offspring produced by breeding and mutation can be improved, by means of problem-dependent optimization algorithms, before placing it into the next generation. In the case of graph partitioning, this can be done by running some local optimization algorithm such as KL to refine the partitions associated with all of the newly created individuals [4, 13, 105]. While this idea can seem interesting at first sight, as it dramatically speeds up the convergence rate of the GA during the first generations, it may impede its efficiency afterward, by considerably reducing genetic diversity towards the norm imposed by the local optimization algorithm, thus making it harder to escape from local optima of the cost function.

In a somewhat different approach, Soper, Walshaw and Cross [111] use the JOSTLE multilevel graph partitioner to compute locally optimal partitions from a set of perturbed instances of the original problem graph, these perturbations taking the form of slight variations in vertex and edge weights. Their crossing-over operator computes a new perturbed graph from two or more of such partitions, by lowering the weight of edges which belong to frontier areas in all of the considered parents, such that the optimized partition which will be computed from the new perturbed graph is likely to have its frontier close to that of its parents. While this approach is reported to compute high quality partitions, and can easily be parallelized according to the multi-deme paradigm, its current sequential implementation is extremely time-consuming.

While the use of refinement methods within a GA framework may not prove a judicious combination, the use of GA as a refinement method within a multilevel framework seemed more promising [72, 105]. Both of the cited methods have been implemented in a sequential way. In [72], the GA refinement method is applied to the whole graphs during the uncoarsening process, resulting in large compute times, while in [105] GA are only applied to groups of 50 to 80 frontier vertices, in order to smooth the partition after a greedy rebalancing phase has taken place.

In the second paper, a method called LRGA is also presented, in which GA refinement takes place only on a small band of vertices around the current separator. In this GA, mutations happen with a higher probability for frontier vertices (where this behavior may be desirable to start the realignment of a whole line of vertices along the faces of a set of mesh elements) than for internal vertices (where this behavior is not likely to bring any improvement). Yet, unlike the band graph method which we will present later, this banded GA refinement method does not take place within a multilevel framework, resulting in worse partitions.

Refinement by diffusion-based methods

Many authors had already noticed that partitions yielded by local optimization algorithms were not optimal. One of the most vocal communities was the users of iterative linear system solving methods [31, 116], who found that such partitions were not fitted for their purpose, because subdomains with longer frontiers or irregular shapes resulted in a larger number of iterations to achieve convergence. To measure the quality of each of the parts, several authors refer to a metric called *aspect ratio*, which can be thought in d dimensions as the ratio between the size of

the interface of a part with respect to the $\left(\frac{d-1}{d}\right)^{th}$ power of the size of its contents[1] [29, 31, 104]. The more compact a part is, the smaller its aspect ratio value is.

In [31], Diekmann *et al.* demonstrated such a behavior, and proposed both a measure of the aspect ratio of the parts, as well as a set of heuristics to create and refine the partitions, with the objective of decreasing their aspect ratio. Among these algorithms is a bubble-growing algorithm, the principle of which was presented in Section 2.5.3. An important drawback of this method is that it does not guarantee that all parts will hold the same number of vertices, which means that afterward other heuristics have to be called on to restore the load balance. Also, all of the graph vertices must be visited many times, which makes this algorithm quite expensive, the more so because it is combined with costly algorithms such as simulated annealing, and the computation of the aspect ratio requires some knowledge on the geometry of the graphs, which is not always available.

In [85], Meyerhenke and Schamberger further explore the bubble model, and devise a way to grow the bubbles by solving, possibly in parallel, systems of linear equations, instead of iteratively computing bubble centers. This method yields partitions of high quality too, but is very slow, even in parallel [86], and the load balancing problem is also not addressed, which means resorting to a greedy load balancing algorithm afterward. Speed concerns were partially resolved by their integration of our band graph technique [84], which we are going to describe in detail in the next section.

4.2 Band graphs

4.2.1 Reducing problem space

Since the strong sequentiality constraint of local refinement algorithms seemed too hard to overcome, resulting either in a loss of partition quality or offering poor scalability, the most straightforward path towards efficient parallel refinement was to try to reduce the cost of global algorithms without losing in partition quality; otherwise, the solutions already implemented in PARMETIS would already have been an acceptable trade-off. This strong constraint on the preservation of quality prevents the weakening of the algorithms in any way. The only solution was therefore to reduce problem space.

By looking back at the essence of the multilevel framework, it soon appeared that, because of the local nature of both the FM and the uncoarsening algorithms, it was most likely that the refined partition computed at any level would not differ much from the partition that was prolonged back to this level, as this latter is itself the prolongation of a partition that was a local optimum in the coarser levels. Therefore, to refine a partition, FM-like algorithms would not need to know more of the graph topology than a small "band" around the boundary of the prolonged partition. We have therefore implemented a band graph extraction algorithm, which only keeps vertices that are at small distance from the prolonged separators, such that our local optimization algorithms are applied to these band graphs rather than to the whole graphs, as illustrated in Figure 4.3 in the sequential case. Vertices which do not belong to the band graph are merged into two *anchor vertices* of weights equivalent to those of the merged vertices.

Using local optimization algorithms such as FM on band graphs substantially differs from what is called "boundary FM" in the literature [53, 68]. This latter technique amounts in FM-like algorithms to recomputing gains of vertices which are in the immediate vicinity of the current separator only, in order to save computation time, and our FM implementations also

[1] For instance, for a 2D mesh, the aspect ratio of a part is measured as the ratio between the length of the perimeter of the part, with respect to the square root of its surface. The lower bound for this metric is achieved for parts of circular shape in the 2D Euclidean space.

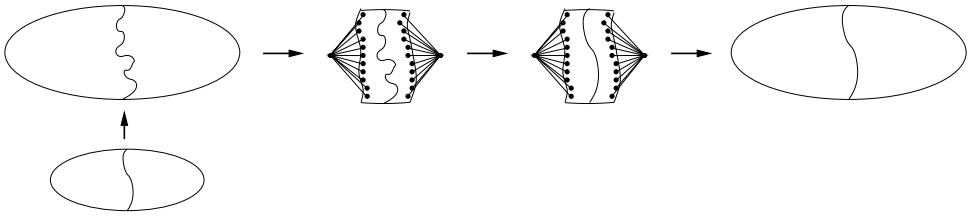


Figure 4.3: Multilevel banded refinement scheme. A band graph of small width is created around the prolonged finer separator, with anchor vertices representing all of the removed vertices in each part. After some optimization algorithm (whether local to the frontier or global to the band graph) is applied, the refined band separator is prolonged back to the full graph, and the uncoarsening process continues.

benefit from this optimization, even on band graphs themselves. What differs with our use of band graphs is that feasible moves are limited to the band area, from which refined separators will never move away, while they could move far away from prolonged separators in the case of unconstrained FM, whether boundary-optimized or not.

4.2.2 Dimensioning and impact of band graphs

Our first experiments with band graphs were carried out on the sequential version of Scotch [23], to refine vertex separators for nested dissection ordering; an edge-oriented version was completed shortly afterward [95], as well as parallel implementations of both of the above [24].

We started by instrumenting our sequential partitioning software Scotch in order to measure how much refined partitions differ from prolonged partitions. Since, at the time, our target application required vertex separators, we focused on them for these experiments, but the same kind of measures could have been obtained from edge separation routines as well. For every separator computed in a nested dissection process (which stops when subgraphs are of sizes of about a hundred vertices), we accumulated the numbers of refined separator vertices that ended up at a given distance from the prolonged separators. These results are presented in Table 4.1.

As expected, the overwhelming majority of refined separator vertices was not located at a distance greater than three from the vertices of the prolonged separators. Therefore, it could be assumed that the quality of partitions should not be impacted if refined partitions are computed on band graphs only. In order to validate this second assumption, we developed in Scotch a partitioning method which extracts a band subgraph of given width from a given graph and its given initial separator, applies an FM separator refinement method to the initial separator of the band subgraph, and prolongs back the refined band separator to the full graph. We then replaced all of our calls to the FM refinement algorithm by calls to this band FM refinement algorithm.

Looking at the results produced, it appeared that band FM refinement seemed to be more stable than the classical FM algorithm. In the former production version of Scotch, two runs of multilevel bipartitioning were performed for each subgraph, and then the best separator of the two was kept. When using band FM refinement, equivalent results were obtained with only one run, as presented in Table 4.2.

Most of the time, the quality of band FM lies between the one exhibited by one and two runs of the classical FM method. In terms of time, we can evidence a moderate over-cost with respect to a single run of classical FM, because of the computation of the band graph. Our interpretation is that pre-constrained banding prevents local optimization algorithms from exploring and being trapped in local optima that would be too far from the global optimum

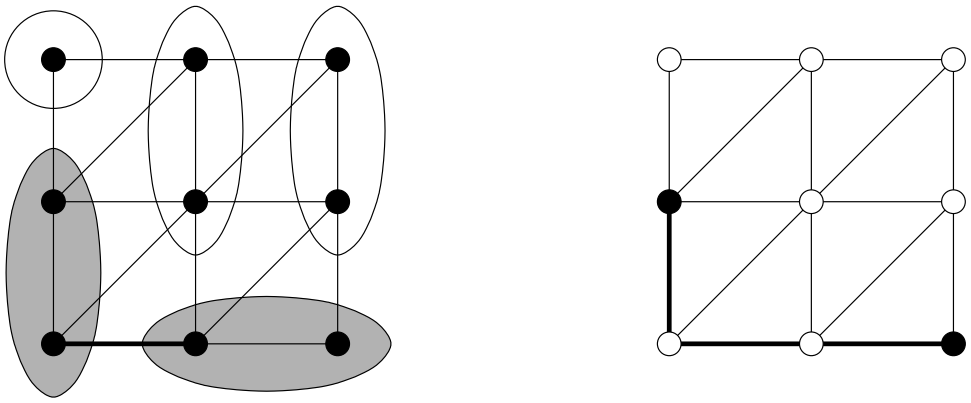


Figure 4.4: When two coarse vertices are at distance 1 from each other, their associated fine vertices are at most at distance 3. Ellipses in the left picture represent coarse vertices.

sketched at the coarsest level of the multilevel process. The optimal band width value of 3 that we have evidenced is significant in this respect: it is the maximum distance at which two vertices can be in some graph when the coarse vertices to which they belong are neighbors in the coarser graph at the next level (see Figure 4.4). Therefore, keeping more layers of vertices in the band graph is not useful, because allowing the fine separator to move a distance greater than 3 in the fine graph to reach some local optimum means that it could already have moved to this local optimum in the coarser graph, save for coarsening artefacts, which are exactly what we do not want to be influenced by. Moreover, even if the separator cannot move more than three vertices away at any level, it has the ability to move again at the next levels to reach its local optimum, therefore compensating on several levels for the necessary moves it could not do on a single level.

4.2.3 Distributed band graphs

The computation and use of distributed band graphs does not differ much from their sequential counterparts [58]. Given a distributed graph and a prolonged separator, which can span across several processes, vertices which are closer to separator vertices than some small user-defined distance are selected, by spreading distance information from all of the separator vertices by means of as many halo exchanges as the desired width of the band graph. Then, the distributed band graph is created, by adding on every process two anchor vertices, which are connected to the last layers of vertices of each of the parts. The vertex weight of the anchor vertices is equal to the sum of the vertex weights of all of the vertices they replace, to preserve the balance of the two band parts.

While sequential band graphs have only one anchor vertex per part, we did not transpose this to the parallel case, because anchor vertices would have had very high degrees, possibly creating communication bottlenecks and interfering with the internals of some optimization algorithms. In the parallel bipartitioning case, there are two anchor vertices per process, so that each of the anchor vertices is connected to all of the local vertices of the last layer belonging to the same part, as well as to all the remote anchor vertices of the same part, forming two distributed cliques, as illustrated in Figure 4.5. Of course, for numbers of processes above tens of thousands, another system should be implemented, for instance some flavor of tree or some low degree, low diameter graph such as the hypercube. As for the sequential case, once the separator of the band graph has been refined using some parallel refinement algorithm, the new separator is prolonged back to the original distributed graph.

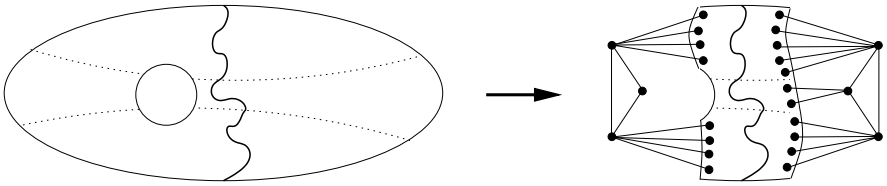


Figure 4.5: Computation of a distributed band graph from a bipartitioned graph distributed across three processes. The solid line is the current partition frontier, while dotted lines represent the separation between process domains. Merged vertices in each part are now represented by a clique of local anchor vertices, one per process and per part.

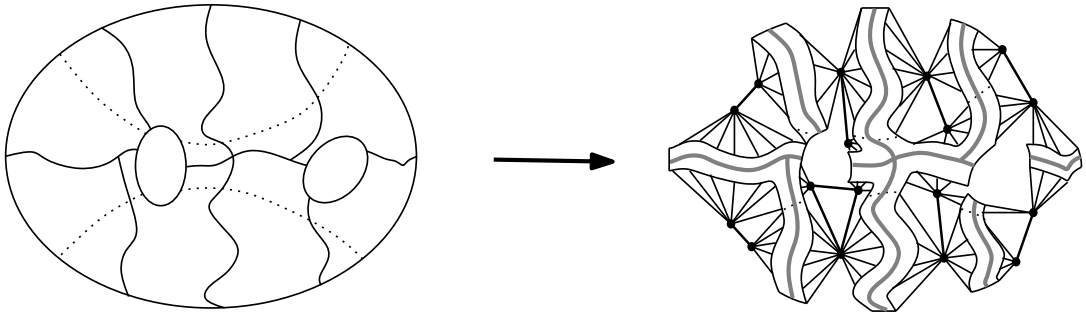


Figure 4.6: Computation of a distributed band graph from a 8-partitioned graph distributed across three processes. The solid line is the current partition frontier, while dotted lines represent the separation between process domains. Merged vertices in each part are now represented by a clique of local anchor vertices, one per process and per part. Anchor vertices which would not be connected to vertices in their part are not created.

At the time being, band graphs extraction methods are available, both for centralized and distributed graphs, for the vertex separation and edge bipartitioning strategies of Scotch. This work is currently being extended to the k-way case, by creating as many cliques of anchor vertices as there are parts in the partition, as illustrated in Figure 4.6. This k-way band graph is planned to be used in conjunction with k-way versions of the algorithms described in sections 4.4.1 and 4.5.3.

Although the distribution of band graph data is likely to be extremely imbalanced, with some processes even being completely idle, to date, we do not redistribute band graph data before calling parallel refinement methods, because we assume that the cost of redistribution would be too high[2]. Yet, when running on very large machines, where collective communication costs are likely to be very high, it might be interesting to fold-dup band graphs on several subsets of processes and to run refinement methods concurrently and independently on each of these subsets. The application to parallel GA of such a multiple fold-dup will be described in Section 4.4.1.

[2]All the more so as we perform multi-centralization on band graphs just after they are created; see next section.

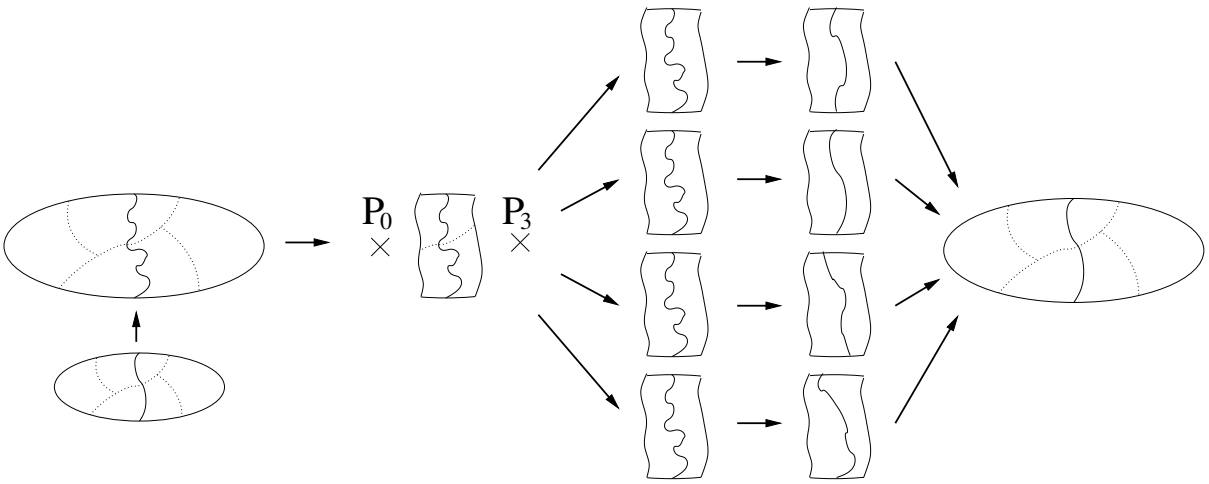


Figure 4.7: Diagram of the multi-sequential refinement of a separator prolonged back from a coarser graph distributed across four processes to its finer distributed graph.

4.3 Multi-centralization

The implementation of distributed band graphs in PT-Scotch was accompanied by the coding of a multi-centralized refinement algorithm, outlined in Figure 4.7. At every distributed uncoarsening step of the multilevel framework, a distributed band graph is created. Centralized copies of this band graph are then gathered on every participating process, which can run fully independent instances of sequential local optimization algorithms such as FM. The perturbation, on every process, of the initial state of the sequential FM algorithm, by permuting the order in which vertices of same gain value are inserted into the gain arrays, allows each of the local optimization algorithms to explore slightly different solution spaces, and thus to improve refinement quality. Finally, the best refined band separator is prolonged back to the distributed graph, and the uncoarsening process continues.

This method is clearly not scalable at all. There will always be a point where centralized graphs will no longer fit in the memory of the processors of the parallel machine, and where the execution of so many local optimization algorithms on the multi-centralized band graphs will be a waste of run time because, in spite of the perturbations of their initial state, most of the obtained local optima will be identical.

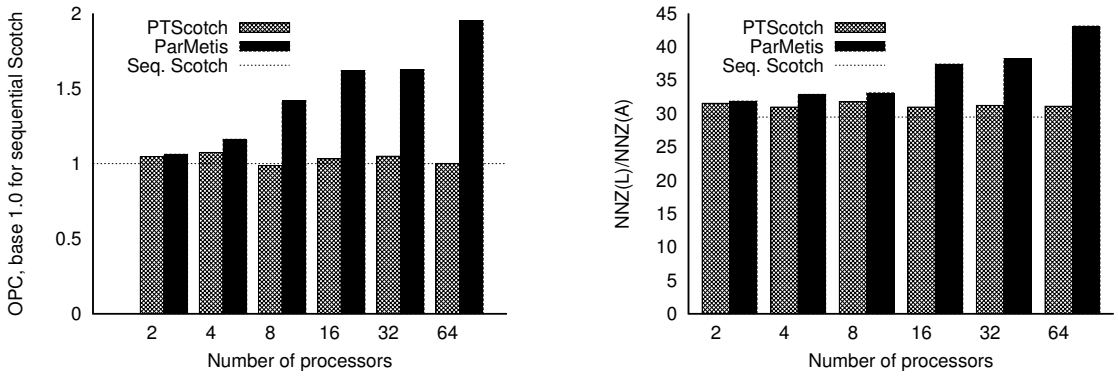
Yet, as a temporary solution, it has proved useful to preserve separator quality when computing nested dissection orderings for graphs of up to 82 million vertices, on hundreds of processors. This was one of the bottlenecks we had to overcome in order to provide a fully parallel software chain, in conjunction with the PaStiX parallel sparse linear solver also developed within our team [36, 57], for solving some large-scale structural mechanics and electro-magnetics problems [48].

The limitation of problem space offered by band graphs seems to be significant enough to allow us to compute high-quality partitions of distributed 3D mesh graphs of up to a billion vertices, which was the goal of our initial roadmap. Since the expected size of the separator of a n-vertex 3D mesh graph is in $O(n^{2/3})$ [110], the order of magnitude of the first separator of a 3D graph about a billion vertices should be of about a million vertices, which can be handled by a sequential computer.

Tables 4.3 and 4.4 present the operation counts (OPC) computed on the orderings yielded by PT-Scotch and ParMeTiS for some test graphs. These results have been obtained by running

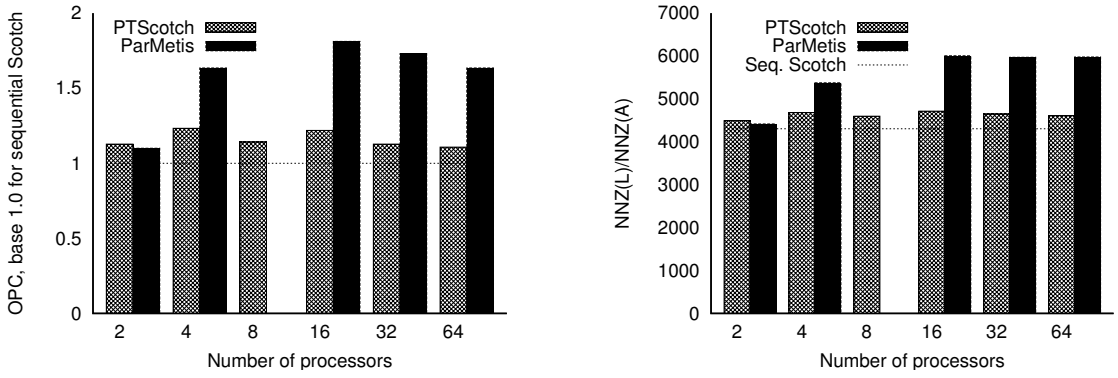
PT-Scotch with the following default strategy: in the multilevel process, graphs are coarsened without any folding until the average number of vertices per process becomes smaller than 100, after which the fold-dup process takes place until all graphs are folded on single processes and the sequential multilevel process relays it.

On these moderate numbers of processes, the improvement in quality of PT-Scotch is clear. Ordering quality does not decrease along with the number of processes, as our local optimization scheme is not sensitive to it, but instead most often slightly increases, because of the increased number of multi-sequential optimization steps which can be run in parallel. Graphs in Figures 4.8 and 4.9 evidence that PT-Scotch clearly outperforms ParMeTiS in term of ordering quality. For both graphs, the results of PT-Scotch are very close to those obtained by the sequential Scotch software, while the costs of ParMeTiS orderings increase dramatically along with the number of processes.



(a) OPC for graph audikw1. (b) NNZ fill ratio for graph audikw1.

Figure 4.8: Ordering results with PT-Scotch and ParMeTiS for graph audikw1. Figures extracted from [24].



(a) OPC for graph cage15. (b) NNZ fill ratio for graph cage15.

Figure 4.9: Ordering results with PT-Scotch and ParMeTiS for graph cage15. Figures extracted from [24].

While PT-Scotch is slower on average than ParMeTiS, because it computes its orderings by means of recursive bipartitioning rather than by extrapolation from a direct k-way partitioning, it can yield operation counts that are as much twice as small as those from ParMeTiS, which is of interest as factorization times are more than one order of magnitude higher than or-

dering times. For example, it took only 41 seconds to version 5.0.6 of PT-Scotch to order the **brgm** matrix on 64 processes, and about 280 seconds to MUMPS [2] to factorize the reordered matrix on the same number of processes. As the factorization process is often very scalable, it can happen, for very large numbers of processes, that ordering times with PT-Scotch are higher than factorization times, because communication latencies dominate and scalability can no longer be guaranteed. Because of their different complexity and scalability behavior, it is in general not useful to run the ordering tool on as many processes as the solver; what matters is to have enough distributed memory to store the graph and run the ordering process, while keeping good scalability in time. Moreover, in various applications, the ordering phase is performed only once while the factorization step is repeated many times with different numerical values, the matrix structure being invariant. This is the reason why, at first, we essentially focused on ordering quality rather than on optimal scalability.

As seen above, the multi-sequential FM refinement algorithm that we currently use is by nature not scalable, and its cost, even on band graphs, is bound to dominate parallel execution time when the number of processors increases. This is why on some of the graphs presented here, depending on their size, average degree and topology, the run time of PT-Scotch no longer decreases. This is why we investigated global refinement methods.

4.4 Genetic algorithms

Local optimization algorithm are not well suited because of their iterative nature, which implies sequential computations on large chunks of the graph so as to benefit from their hill-climbing capabilities on the enclosed portions of the frontier. As evidenced in [119], the trade-off between scalability and partition quality lies in the number and size of the chunks. On the contrary, global heuristics, although potentially more scalable, are usually not considered as good candidates because of the size of the problem spaces to explore. However, the band graphs which we introduced in the previous section result in a dramatic reduction of problem space. Performing sequential optimization on band graphs is somewhat equivalent to applying the method of Walshaw and Cross to only one big chunk containing all of the area around the frontier, which is obviously not scalable in their case. However, global methods could well take advantage of this reduction.

The first class of global methods that we considered for this purpose was genetic algorithms [23]. The proof-of-concept algorithm which we developed in order to validate the use of genetic algorithms on band graphs was based on the sequential version of Scotch, to speed-up developments, and used thread-based parallelism to implement concurrency.

Since, at that time, we were focusing on sparse matrix ordering, individuals of the genetic population represent vertex separators[3]. In the graph separation problem, every vertex can belong to three different domains: the separator, or any of the two separated parts. Therefore, every individual in the population is represented as a linear array, modeling a chromosome, which associates a number between 0 and 2 to any graph vertex index.

The reproduction operator is a classical multi-points cross-over operator, which is applied at a randomly-selected position of two mated individuals, and swaps one part of their arrays to produce two descendants. The mutation operator consists in swapping the part of randomly chosen vertices on some individuals. Since these naive operators cannot enforce that the crossed-over and mutated individuals be valid solutions, they are post-processed with a consistency-checking phase which adds vertices to the separator whenever necessary, and removes unneeded

[3]However, the solution which we implemented easily transposes to the edge partitioning case, in a way similar to the one presented in [113].

separator vertices.

Individuals are evaluated by means of a fitness function, which linearly combines dimensionless numbers such as the ratio of graph vertices that belong to the separator, the imbalance between the two parts, and the ratio of graph edges that link separator vertices. The first generation is made up of individuals that are mutations of the prolonged partition, plus some entirely random individuals which provide genetic diversity. To select and mate individuals, we have implemented several classical algorithms [61, 72, 88]. Although all methods behave quite similarly, best results were achieved in our case with a mix of the elitism and roulette methods: the 5% best individuals are kept unconditionally, and each of the remaining ones is kept with a probability proportional to its fitness; the efficiency of elitist policies in the case of graph partitioning has also been noted by other authors [104, 111], as related to the fact that most of the offspring generated are not of very high quality. Then, individuals are mated by pairs of descending fitness, and bred so as to keep constant population. Our prototype sequential implementation was multi-deme, each of the demes being handled by a different thread. Migration is performed when the variety of the population in some deme decreases, *i.e.* when individuals are too similar to their local champion.

To evaluate the convergence speed of our GA algorithm, we computed nested dissection orderings of several test graphs with our multilevel band GA method. These tests were run on the M3PEC machine of the Université Bordeaux I, an eleven-node IBM machine with eight 1.5 MHz dual-core processors and 32 GB of memory per node. Since our implementation was thread-based only, timings of tests involving more than sixteen threads (written between parentheses) are estimated: these tests are still run on a single SMP node, with as many threads per core as necessary, and the running time is divided by the appropriate ratio. PARMETIS, however, uses MPI, and runs fully in parallel.

Table 4.5 presents some results for graph BCSSTK29. These results show that GA converges quite well, and that quality can be improved by increasing computation time and/or population size. As expected, running times are high, but GA are highly scalable, so that computation time can be reduced by adding processes, and partitioning quality can be increased by giving more time. The comparison between the two lines of equivalent computational cost, that is, $(80, 1, 25)$ and $(40, 2, 25)$, is also very informative, as it shows the interest of multiple demes when mating is performed on an elitism basis. Indeed, the problem with the elitism selection policy is that it might wipe out individuals which are too far from the optimum while possessing characteristics which, combined with the best individuals, would yield an improvement. Being able to preserve such individuals on separate demes preserves genetic diversity. In the above cases, more demes with smaller populations mean faster execution (because of increased concurrency in the GA method) and better solution (because of increased genetic diversity).

The second class of experiments that we have run aimed at evaluating the scalability of our method in terms of quality and running time. In order to compare our ordering software to PARMETIS in similar conditions, we ran our method on numbers of processors p which are powers of two[4] and performed band GA on the first $\log_2(p)$ levels only, using band FM afterward. This execution scheme, which we will refer to as "limited GA" (LGA) in all of the following, aims at simulating the execution of PT-SCOTCH on p processes, where distributed algorithms (including some distributed version of our GA method) would be used on the first $\log_2(p)$ levels of nested dissection, before the software switches to purely sequential processing which can make use of the sequential FM local optimization method. When running the threaded GA method, the population is evenly spread on all of the threads, with at least 100 individuals on

[4] PARMETIS can only compute orderings on numbers of processes which are a power of two. Our method itself bears no such limitation.

the whole and at least 25 individuals per deme; therefore, from 4 threads and above, the global population remains constant per thread and doubles with the number of threads.

Our results, which are summarized in Table 4.6, are mixed, but encouraging as our implementation left much room for improvements. On the whole, partitioning quality tends to degrade when the number of processors increases: on our worst case, BMW32, we lose about 60% in OPC quality between 1 and 64 processors, while it remains almost constant for COUPOLE8000. However, above 8 processors, our results clearly outperform those of PARMETIS, by a factor greater than two for THREAD. As expected, the higher the degree of the graph is, the bigger the difference, because PARMETIS can only do gradient local optimizations on nodes which have neighbors on other processors. In that respect, GA prove they can offer a scalable solution to the problem experienced by parallel implementations of FM-like algorithms. Yes, the loss in separator quality has to be fought.

Partitioning times are also promising. Although the run time of a single sequential band GA refinement algorithm is between 30 and 80 times higher than that of its sequential band FM counterpart, the scalability of GA allows it to compete favorably with distributed FM-like methods for large numbers of processes. In the above results, the overall running time of our LGA ordering program does not increase much with respect to FM-like methods when the number of processors increase. While a doubling of the number of processors implies the turning of a whole level of band FM refinements into band GA refinements, it is only smaller and smaller subgraphs which are passed to GA instead of FM, resulting in a bounded increase of run time.

4.4.1 Parallelization of genetic algorithms

For reasons which will be exposed in the next section, the prototype implementation described above has not been turned into a distributed, fully functional production code. Yet, we are convinced that, when applied to band graphs and when judiciously accelerated by means of more specialized methods, distributed GA can be an effective algorithm for parallel partition refinement. However, in order to achieve maximum scalability, its implementation must possess several key features.

First, it should be able to handle chromosomes split across several processes. For graphs well above a billion of vertices, centralized band graphs will no longer hold in the memory of compute nodes, and storing the individuals as well as computing their fitness must consequently be performed in a distributed way, based on the assignment of the vertices of the distributed band graph to the nodes of the target machine. Second, in order to preserve the desirable multi-deme feature, demes should also be able to span across several processes. However, in order to localize intra-deme communication, as well as inter-deme communication when exchanging champions, demes should span on as few processes as possible. The number of processes onto which chromosomes will have to be split can be estimated depending on available memory, on the size of the band graph, and on the desired maximum population per deme. This will give an upper bound on the number of demes. Then, knowing the number of desired or possible demes and/or the number of processes per deme, band graphs can be redistributed such that each of the demes will possess a copy of the (re-)distributed band graph[5]. The distributed GA computation can then take place, synchronously or asynchronously, on each of the sub-communicators created for each of the demes, with the occasional sending of champions from the processes of one deme to their counterparts of some other deme. The potential asynchronicity of the method might

[5]This redistribution will in fact be a concentration, since the data of band graphs distributed across all of the available processes will be gathered onto limited subsets of these processes, in multiple copies. This process is very much like a fold-dup, albeit in multiple copies and not just two.

well be its key advantage on very large target architectures of the future, compared to the method described below.

4.5 Diffusion algorithms

As demonstrated in the previous section, GAs have interesting characteristics in terms of scalability and quality. However, their convergence rate is slow, because they are not specialized enough with respect to the problem at stake, as illustrated in Figure 4.2. In order to save time and compute power, it appeared worthwhile to turn to methods which specifically address our requirements for small and smooth frontiers, while still being of global and scalable nature. This is why we investigated diffusion-based methods instead of developing a fully parallel version of our GA method.

In spite of the quality of the partitions they provide, all of the existing diffusion schemes (see Section 4.1.2) had two drawbacks. First, they did not intrinsically balance loads between parts, which required further post-processing of the produced partition by local migration algorithms of an iterative nature, that is, just those which we wanted to eliminate; second, they were quite expensive, as they involved all of the graph vertices. The method which we devised [95], and which is described below, aimed at addressing concurrently both of these issues.

4.5.1 The jug of the Danaides

Our diffusion scheme can apply to an arbitrary number of parts, but for the sake of clarity we will describe it in the context of graph bipartitioning, that is, with two parts only. We modeled the graph to bipartition in the following way, depicted in Figure 4.10. Nodes are represented as barrels of infinite capacity, which leak so that one unit of liquid at most drips per unit of time. When graph vertices are weighted, always with integer weights, the maximum quantity of liquid to be lost per unit of time is equal to the weight of the vertex. Graph edges are modeled by pipes of cross-section equal to their weight. In both parts, a source vertex is chosen, to which a source pipe is connected, which flows in $\frac{|V|}{2}$ units of liquid per unit of time. Two sorts of liquids are in fact injected in the system: scotch in the first pipe, and anti-scotch in the second pipe, such that when some quantity of scotch mixes with the same quantity of anti-scotch, both vanish[6]. To ease the writing of the algorithm in the bipartitioning case, scotch is represented by positive quantities and anti-scotch is represented by negative ones, so that mutual destruction naturally takes place when adding any two quantities of opposite signs.

The diffusion algorithm performs as outlined in Figure 4.11. For each time step, and for each vertex, the amount of liquid (whether scotch or anti-scotch) which remains after some has leaked is spread across the connecting pipes towards the neighboring barrels, according to their respective sections. This process could be iterated until convergence, but in fact it is only performed for a number of steps sufficient to achieve sign stability. Indeed, we are not interested in complete convergence, but in the stability of the signs of all content quantities borne by graph vertices, which indicate whether scotch or anti-scotch dominates in the barrels, that is, if the vertex belongs to part 0 or 1.

Since $|V|$ units of both liquids are injected on the whole per unit of time, and since all of the barrels can leak the same overall amount in the same time, the system is bound to converge, all the more so that liquid can disappear by collision of scotch and anti-scotch. As in the bubble schemes, what is expected is that a smooth front will be created between the two parts.

[6]The author cannot imagine what "anti-scotch" can be, and therefore has not provided an explicit name for it. The idea of having so much scotch leak from its barrels, or to have it vanish, is also quite revolting for any civilized mind. Science indeed imposes sacrifices.

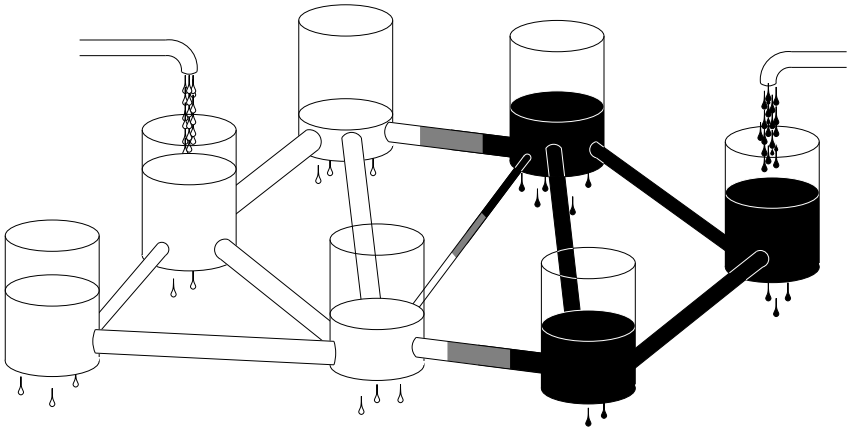


Figure 4.10: Sketch of our diffusion model.

The purpose of the algorithm is more to have a global smoothing of the frontier than a strict minimization of the cut. In fact, unlike all of the algorithms presented in Section 4.1.2, our method privileges load balancing over cut minimization. For this latter criterion, we rely on an additional feature of our scheme, as explained below.

4.5.2 Diffusion on band graphs

Our diffusion algorithm, as such, presents two weaknesses: nothing is said about the selection of the seed vertices, and performing such iterations over all of the graph vertices is very expensive compared to local optimization algorithms which only consider vertices in the immediate vicinity of the frontiers. However, both issues can be solved by the use of the band graphs presented in Section 4.2.

Anchor vertices represent a natural choice to be taken as seed vertices. Indeed, the most important problem for bubble-growing algorithms is the determination of the seed vertices from which bubbles are grown, which requires expensive processes involving all of the graph vertices [31, 85]. Since anchor vertices are connected to all of the vertices of the last layers, the diffused liquids flow as a front as if they originated from the farthest vertices from the frontier, which is indeed what would happen if they flowed from the center of a bubble having the frontier as its perimeter.

The diffusion algorithm discussed above has been implemented as a sequential graph bipartitioning method. All of the necessary floating-point arithmetic has been implemented in single precision.

These tests were run on a Lenovo ThinkPad T60 laptop, with an Intel dual-core T2400 processor running at 1.8 MHz and 1 Gb of memory. As we ran sequential tests only, the dual-core feature of the processor is not relevant. Test graphs were partitioned into 2 to 128 parts, and three quality metrics were considered: the number of cut edges, called **Cut**; a load imbalance ratio equal to the size of the largest part divided by the average size, called **MaCut**; and the maximum diameter of the parts, referred to as **MDi**, which is an indirect metric of the shape of the partition, and is usable even in the case of graphs of unknown or nonexistent geometry. The latter metric is however insufficient, as it does not really capture the smoothness of the interfaces, since irregularly shaped parts can still have small diameters.

Three diffusion heuristics were compared against the classical multilevel recursive bipar-

```
while (number of passes to do) {
  reset contents of new array to 0;
  old[s_0] ← old[s_0] − |V|/2;          /* Refill source barrels */
  old[s_1] ← old[s_1] + |V|/2;
  for (all vertices v in graph) {
    c ← old[v];                         /* Get contents of barrel            */
    if (|c| > weight[v]) {              /* If not all contents have leaked   */
      c ← c − weight[v] * sign(c);      /* Compute what will remain          */
      σ ← Σ_{e=(v,v')} weight[e];       /* Sum weights of all adjacent edges */
      for (all edges e = (v, v')) {     /* For all edges adjacent to v       */
        f ← c * weight[e]/σ;            /* Fraction to be spread to v'       */
        new[v'] ← new[v'] + f;          /* Accumulate spreaded contributions */
      }
    }
  }
  swap old and new arrays;
}
```

Figure 4.11: Sketch of the jug of the Danaides diffusion algorithm. Scotch, represented as positive quantities, flows from the source of part 1, while anti-scotch, represented as negative quantities, flows from the source of part 0. For each step, the current and new contents of every vertex are stored in arrays `old` and `new`, respectively.

titioning strategy implemented in Scotch 4.0, referred to as RMF in the following, which performs recursive bipartitioning with bipartitions computed in a multilevel way, using FM refinement.

The first method, RMBD, uses the same recursive bipartitioning and multilevel strategy, but banded diffusion is performed during the multilevel refinement steps. The results achieved with this method validate our approach: the obtained partitions have very smooth boundaries (see Figure 4.13.b), and are adequately balanced if the number of diffusion iterations is sufficiently high, as shown in Table 4.7.

When performing 100 diffusion steps, the average **MaCut** value for RMBD is 1.046, only 1.80 % higher than the one of RMF. However, the maximum diameter **Mdi** is not significantly reduced, and is even increased on average by 4.69% with respect to RMF. This method is also 5.33 times slower than RMF and increases the cut by about 20%, which makes it of little practical use.

We have therefore experimented a second method, RMBDF, where the classical FM algorithm is applied to the band graph after the diffusion algorithm. The idea of this strategy is to benefit from the global optimization capabilities brought by the diffusion algorithm, while locally optimizing the frontier afterward. Even when performing 40 diffusion steps only, the smoothness of the boundaries is preserved and parts are more balanced, while the cut is only increased by 3.10% with respect to RMF. This strategy is also only three times slower than RMF, which is extremely fast for a diffusion-based algorithm.

In order to favor the minimization of diameters, we have modified our diffusion method so as to double at each step the amount of liquid borne by every vertex, in an "avalanche"-like process. This method is referred to as "aD". It is no longer bound to converge, and indeed causes overflows for large numbers of diffusion steps, but gives good results for small numbers of iterations. As a matter of fact, we can see in Table 4.7 that the RMBaDF method is the most efficient one on average, and yields better results than the classical RMF method while still providing smooth boundaries, as evidenced in Figure 4.13.c. Further studies are however necessary to determine whether this scheme is stable enough for larger band graphs or yields overflows before all signs have been stabilized.

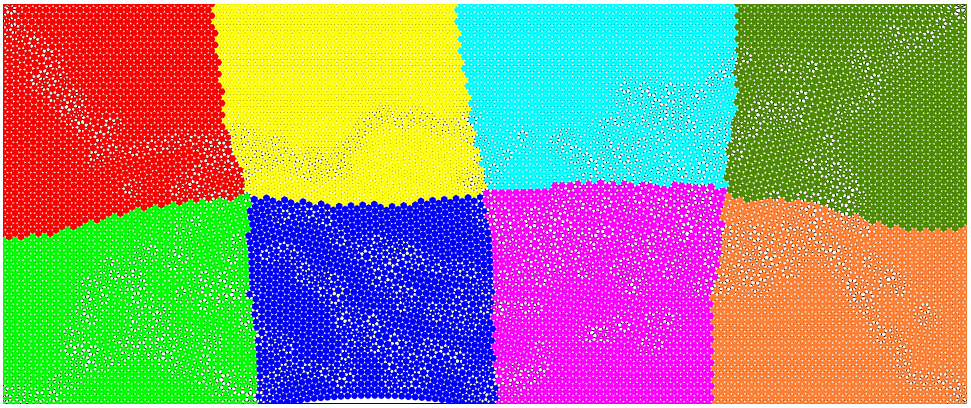
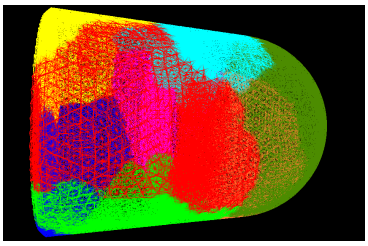
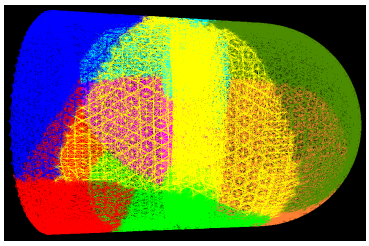


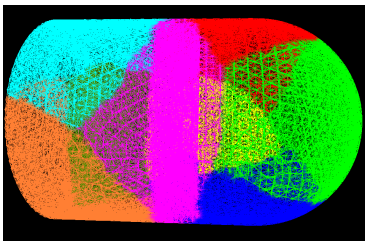
Figure 4.12: Partition of graph BUMP into 8 parts with SCOTCH 5.0.6, using the MBDF strategy. The cut is equal to 713 edges. This picture is to be compared with Figure 2.8, page 15.



(a) RMF.



(b) RMBD.



(c) RMBaDF.

Figure 4.13: Partition of graph ALTR4 into 8 parts using SCOTCH 5.0.6 with three different strategies. The segmented frontiers produced by FM-like algorithms are clearly evidenced in Figure (a). RMBD produces the smoothest boundaries, as shown in Figure (b). RMBaDF takes the best of both worlds, in Figure (c).

Table 4.8 compares some of our results against those obtained with KMETIS. KMETIS uses direct k-way partitioning instead of recursive bipartitioning, which usually makes it more efficient when the number of parts increases, and also much faster (from 10 to 20 times). As analyzed in [110], the performance of recursive bipartitioning methods tends to decrease when the number of parts increases, which should limit the efficiency of RMBDF methods for large numbers of parts. A full k-way diffusion algorithm is therefore required.

4.5.3 Parallelization of the diffusive algorithms

Our diffusion algorithm has the additional benefit of being highly scalable, and its parallel version does not substantially differ from its sequential counterpart [58]. It relies on the low-level halo exchange routines of the PT-SCOTCH library to propagate on local ghost nodes the amount of liquids to be spread, per edge weight unit, by each non-local neighbor vertex as the result of the previous iteration, and uses these values to compute the new amounts for all of the local vertices, including the leakage effect once all contributions have been received.

As said above, to date, it only handles two kinds of liquid, limiting its use to the recursive bipartitioning case.

We present below the results that we have obtained with the most recent version of PT-Scotch by the time this dissertation is written, that is, revision 5.1.6. This revision includes all of the contributions that were discussed in the previous sections. In these tests, PT-Scotch has been used to compute graph partitions in parallel by means of recursive bisection, using probabilistic matching, fold-dup on the final coarsening stages, and diffusion on distributed band graphs.

Our experiments were performed on the Platine supercomputer at CCRT. This machine is a Bull Novascale cluster of 932 compute nodes interconnected via an Infiniband network. Each node has four dual-core Intel Itanium II processors. The metric that we considered for evaluating the quality of partitions is cut size.

Graph	Distance				
	0	1	2	3	≥ 4
3ELT	53.51	34.23	8.67	2.31	1.29
4ELT2	51.66	37.98	7.33	1.63	1.40
4ELT	53.16	36.29	7.16	1.84	1.54
598A	76.23	23.45	0.32	0.00	0.00
AATKEN	77.00	20.45	2.27	0.24	0.04
ALTR4	74.19	24.89	0.80	0.12	0.00
AUDIKW1	91.44	8.55	0.01	0.00	0.00
AUTO	77.89	21.89	0.22	0.00	0.00
B5TUER	74.18	22.96	1.85	0.42	0.59
BCSSTK29	82.04	17.66	0.30	0.00	0.00
BCSSTK30	87.17	12.53	0.29	0.01	0.00
BCSSTK31	81.90	17.69	0.39	0.02	0.00
BCSSTK32	81.91	17.80	0.23	0.03	0.02
BMW32	80.98	18.31	0.50	0.08	0.14
BMWCRA1	91.29	8.58	0.13	0.00	0.00
BODY	67.49	30.20	2.08	0.20	0.04
BRACKET	72.47	26.19	1.08	0.16	0.10
BUMP	48.33	37.08	9.54	2.73	2.32
CHANEL1M	74.65	24.09	1.16	0.10	0.00
CONESPHERE1M	82.16	17.67	0.17	0.00	0.00
COUPOLE8000	90.23	9.74	0.03	0.00	0.00
CRANKSEG2	95.80	4.17	0.01	0.02	0.00
GUPTA1	96.05	3.61	0.34	0.00	0.00
INLINE1	87.57	12.35	0.08	0.00	0.00
M14B	78.65	21.17	0.18	0.00	0.00
MT1	84.79	14.00	0.93	0.25	0.04
OCEAN	60.43	32.86	4.58	1.29	0.84
OILPAN	77.60	20.54	1.20	0.17	0.49
PWT	54.35	37.31	6.10	1.56	0.69
ROTOR	77.09	21.99	0.75	0.11	0.06
S3DKQ4M2	78.72	20.34	0.89	0.04	0.00
SHIP001	91.43	8.51	0.05	0.00	0.00
SHIPSEC5	82.29	17.28	0.41	0.03	0.00
SPHERE	46.32	39.82	9.44	2.02	2.40
THREAD	91.40	8.53	0.06	0.00	0.00
TOOTH	69.90	26.82	2.42	0.63	0.24
X104	86.64	12.81	0.51	0.03	0.00

Table 4.1: Distance histogram (in % of the number of separator vertices) of the location of refined separator vertices with respect to prolonged separators, for a representative subset of our test graphs. These statistics have been collected over all separators when performing nested dissection on the given graphs. Data extracted from [23].

Graph	Band FM (1 run)		FM (2 runs)		FM (1 run)	
	OPC	Time (s)	OPC	Time (s)	OPC	Time (s)
aatken	1.72e+11	6.17	**1.70e+11**	10.79	1.73e+11	5.38
auto	5.14e+11	47.09	**4.98e+11**	75.00	5.27e+11	39.40
bcsstk32	1.40e+9	1.16	**1.28e+9**	1.65	1.40e+9	1.02
coupole8000	7.57e+10	210.15	**7.48e+10**	346.81	7.57e+10	183.72
m14b	6.27e+10	21.4	6.31e+10	33.42	**6.03e+10**	17.56
tooth	**6.50e+9**	5.66	6.51e+9	9.01	6.71e+9	4.64
audikw1	5.58e+12	59.32	**5.48e+12**	86.78	5.64e+12	50.33
bmw32	3.15e+10	4.52	**2.75e+10**	6.51	3.07e+10	4.08
oilpan	2.92e+9	0.73	**2.74e+9**	0.95	2.99e+9	0.69
thread	4.17e+10	1.62	**4.14e+10**	2.30	4.17e+10	1.44
x104	1.84e+10	1.97	**1.64e+10**	2.60	1.80e+10	1.83
altr4	3.68e+8	1.55	**3.65e+8**	2.52	3.84e+8	1.32
conesphere1m	**1.83e+12**	122.03	1.85e+12	192.27	1.88e+12	100.19

Table 4.2: Comparison between band FM and classical FM. Tests have been run on a 375MHz IBM SP3.

Test case	Number of processes					
	2	4	8	16	32	64
23millions						
O_{PTS}	**1.45e+14**	**2.91e+14**	**3.99e+14**	**2.71e+14**	**1.94e+14**	**2.45e+14**
O_{PM}	†	†	†	†	†	†
t_{PTS}	671.60	416.75	295.38	211.68	147.35	103.73
t_{PM}	†	†	†	†	†	†
altr4						
O_{PTS}	**3.84e+8**	**3.75e+8**	**3.93e+8**	**3.69e+8**	**4.09e+8**	**4.15e+8**
O_{PM}	4.20e+8	4.49e+8	4.46e+8	4.64e+8	5.03e+8	5.16e+8
t_{PTS}	0.42	0.30	0.24	0.30	0.52	1.55
t_{PM}	0.31	0.20	0.13	0.11	0.13	0.33
audikw1						
O_{PTS}	**5.73e+12**	**5.65e+12**	**5.54e+12**	**5.45e+12**	**5.45e+12**	**5.45e+12**
O_{PM}	5.82e+12	6.37e+12	7.78e+12	8.88e+12	8.91e+12	1.07e+13
t_{PTS}	73.11	53.19	45.19	33.83	24.74	18.16
t_{PM}	32.69	23.09	17.15	9.804	5.65	3.82
bmw32						
O_{PTS}	3.50e+10	**3.49e+10**	**3.14e+10**	**3.05e+10**	**3.02e+10**	**3.00e+10**
O_{PM}	**3.22e+10**	4.09e+10	5.11e+10	5.61e+10	5.74e+10	6.31e+10
t_{PTS}	8.89	7.41	5.68	5.45	8.36	17.64
t_{PM}	3.39	2.28	1.51	0.92	0.68	1.08
brgm						
O_{PTS}	**2.70e+13**	**2.55e+13**	**2.65e+13**	**2.88e+13**	**2.86e+13**	**2.87e+13**
O_{PM}	–	†	†	†	†	†
t_{PTS}	276.9	167.26	97.69	61.65	42.85	41.00
t_{PM}	–	†	†	†	†	†
cage15						
O_{PTS}	4.58e+16	**5.01e+16**	**4.64e+16**	**4.95e+16**	**4.58e+16**	**4.50e+16**
O_{PM}	**4.47e+16**	6.64e+16	†	7.36e+16	7.03e+16	6.64e+16
t_{PTS}	540.46	427.38	371.70	340.78	351.38	380.69
t_{PM}	195.93	117.77	†	40.30	22.56	17.83
conesphere1m						
O_{PTS}	**1.88e+12**	**1.89e+12**	**1.85e+12**	**1.84e+12**	**1.86e+12**	**1.77e+12**
O_{PM}	2.20e+12	2.46e+12	2.78e+12	2.96e+12	2.99e+12	3.29e+12
t_{PTS}	31.34	20.41	18.76	18.37	25.80	92.47
t_{PM}	22.40	11.98	6.75	3.89	2.28	1.87

Table 4.3: Comparison between PT-Scotch 5.0.6 (PTS) and ParMeTiS (PM) for several graphs. O_{PTS} and O_{PM} are the operation counts for PTS and PM, respectively, while t_{PTS} and t_{PM} are their run times, in seconds. Dashes indicate abortion due to memory shortage. Daggers indicate abortion due to an invalid MPI operation. Data extracted from [24].

Test case	Number of processes					
	2	4	8	16	32	64
coupole8000						
O_{PTS}	**8.68e+10**	**8.54e+10**	8.38e+10	**8.03e+10**	**8.26e+10**	**8.21e+10**
O_{PM}	†	†	**8.17e+10**	8.26e+10	8.58e+10	8.71e+10
t_{PTS}	114.41	116.83	85.80	60.23	41.60	28.10
t_{PM}	63.44	37.50	20.01	10.81	5.88	3.14
qimonda07						
O_{PTS}	–	–	**5.80e+10**	**6.38e+10**	**6.94e+10**	**7.70e+10**
O_{PM}	†	†	†	†	†	†
t_{PTS}	–	–	34.68	22.23	17.30	16.62
t_{PM}	†	†	†	†	†	†
thread						
O_{PTS}	**3.52e+10**	**4.31e+10**	**4.13e+10**	**4.06e+10**	**4.06e+10**	**4.50e+10**
O_{PM}	3.98e+10	6.60e+10	1.03e+11	1.24e+11	1.53e+11	–
t_{PTS}	3.66	3.61	3.30	3.65	5.68	11.16
t_{PM}	1.25	1.05	0.68	0.51	0.40	–

Table 4.4: Continuation of Table 4.3.

Deme size	# Demes	Generations	OPC	Time (s)
40	1	25	5.32e+08	4.05
80	1	25	5.37e+08	7.95
80	1	100	4.36e+08	25.72
40	2	25	4.65e+08	6.61
40	2	100	4.57e+08	20.17
80	8	100	3.75e+08	50.90

Table 4.5: OPC of the reordered BCSSTK29 matrix when multilevel band GA is used for all levels of nested dissection. Classical multilevel FM yields an OPC of $3.43e+08$ in 0.74s. Data extracted from [23].

Test case	Number of processors or threads						
	1	2	4	8	16	32	64
BCSSTK32							
C_{LGA}	1.60e+9	1.55e+9	1.67e+9	**1.82e+9**	**1.83e+9**	**1.53e+9**	**2.07e+9**
C_{PM}	**1.29e+9**	**1.55e+9**	**1.62e+9**	3.09e+9	4.11e+9	5.85e+9	4.01e+9
t_{LGA}	0.42	0.88	0.84	0.97	2.07	(2.86)	(4.06)
AUDIKW1							
C_{LGA}	**5.68e+12**	**5.91e+12**	**5.70e+12**	**5.82e+12**	**5.99e+12**	**6.44e+12**	**6.02e+12**
C_{PM}	–	–	–	7.78e+12	8.88e+12	8.91e+12	1.07e+13
t_{LGA}	19.78	22.77	29.55	32.89	60.24	(74.64)	(91.78)
BMW32							
C_{LGA}	3.04e+10	3.44e+10	**3.75e+10**	**4.13e+10**	**4.64e+10**	**4.57e+10**	**5.01e+10**
C_{PM}	**2.84e+10**	**3.22e+10**	4.09e+10	5.11e+10	5.61e+10	5.74e+10	6.31e+10
t_{LGA}	1.69	1.79	2.48	2.36	3.67	(5.11)	(7.80)
ALTR4							
C_{LGA}	**3.46e+8**	**3.71e+8**	**4.23e+8**	**4.06e+8**	**4.31e+8**	**4.92e+8**	**4.71e+8**
C_{PM}	4.25e+8	4.20e+8	4.49e+8	4.46e+8	4.64e+8	5.03e+8	5.16e+8
t_{LGA}	0.65	1.78	2.25	1.95	3.36	(5.43)	(7.20)
t_{PM}	0.58	0.31	0.20	0.13	0.11	0.27	0.31
CONESPHERE1M							
C_{LGA}	**1.90e+12**	**1.92e+12**	**1.99e+12**	**2.37e+12**	**2.34e+12**	**2.53e+12**	**2.63e+12**
C_{PM}	2.04e+12	2.20e+12	2.46e+12	2.78e+12	2.96e+12	2.99e+12	3.29e+12
t_{LGA}	44.03	69.66	86.47	90.44	120.87	(134.85)	(158.07)
COUPOLE8000							
C_{LGA}	**7.64e+10**	**7.64e+10**	**7.62e+10**	**7.65e+10**	**7.66e+10**	**7.68e+10**	**7.66e+10**
C_{PM}	–	–	–	8.17e+10	8.26e+10	8.58e+10	8.71e+10
t_{LGA}	125.69	75.40	55.19	49.16	52.59	(61.93)	(77.26)
THREAD							
C_{LGA}	4.10e+10	3.99e+10	**4.41e+10**	**4.64e+10**	**4.43e+10**	**4.59e+10**	**5.19e+10**
C_{PM}	**3.65e+10**	**3.98e+10**	6.60e+10	1.03e+11	1.24e+11	1.53e+11	1.28e+11
t_{LGA}	0.56	2.33	3.10	2.93	4.22	(5.02)	(5.92)

Table 4.6: Comparison between PARMETIS (PM) and our SCOTCH LGA scheme for several graphs. C_{LGA} and C_{PM} are the OPC for LGA and PM, respectively. Dashes indicate abortion due to memory shortage. LGA timings between parentheses are extrapolated times for cases requiring more than 16 threads, as we had to run several threads per core on a single SMP node. Timings for PARMETIS are provided for graph ALTR4 to give an idea of its speed, but t_{PM} and t_{LGA} cannot be compared, because PM is a fully parallel program, while our LGA testbed is the purely sequential nested dissection routine of SCOTCH. Data extracted from [23].

Table 4.7: Evolution of the cut size (**ΔCut**), of the load imbalance ratio (**ΔMaCut**) and of the maximum diameter of the parts (**ΔMDi**) produced by various partitioning heuristics with respect to the RMF strategy, averaged over all test graphs and numbers of parts. Figures below partitioning strategy names indicate the number of diffusion steps performed. Data extracted from [95].

Method	RMBD				RMBDF		RMBaDF
	500	200	100	40	500	40	40
ΔCut (%)	+19.51	+20.01	+18.15	+21.49	+2.26	+3.10	-3.17
ΔMaCut (%)	+0.58	+1.12	+1.80	+9.76	-0.95	-0.29	-0.21
ΔMDi (%)	+3.86	+1.92	+4.69	+5.43	+2.26	+3.10	-3.24
ΔTime (×)	21.31	9.33	5.33	2.93	21.47	2.99	3.07

Table 4.8: Comparison of the results, in terms of cut size (**Cut**) and maximum diameter of the parts (**MDi**), between three heuristics: multi-level with FM refinement (RMF, as implemented in Scotch 4.0), multi-level with banded diffusion and FM refinements (RMBaDF), and kMeTiS. Data extracted from [95].

Test case		**Number of parts**						
		2	4	8	16	32	64	128
altr4								
RMF	**Cut**	1688	**3197**	**4978**	**7788**	**11905**	17656	24478
	MDi	50	52	**40**	33	**25**	21	**14**
RMBaD(40)F	**Cut**	**1621**	3203	5017	7776	11980	17669	24831
	MDi	**48**	46	41	**30**	**25**	**18**	**14**
KMeTiS	**Cut**	1670	3233	4981	8115	12147	**17355**	**24058**
	MDi	**48**	**45**	41	34	26	22	**14**
bmw32								
RMF	**Cut**	17271	**54424**	84222	**120828**	**181844**	**267427**	**394418**
	MDi	93	116	130	106	74	120	68
RMBaD(40)F	**Cut**	16032	54446	**83422**	124945	183454	275594	411154
	MDi	91	130	**96**	**84**	**68**	63	**56**
KMeTiS	**Cut**	**15529**	55506	92658	125686	193169	286111	420965
	MDi	**87**	**108**	99	87	70	**61**	68

Test case	Number of processors:Number of parts									
	32:2	32:32	32:1024	128:2	128:128	128:1024	384:2	384:256	384:1024	P_{peak}:2
10MILLIONS										
C_{PTS}	**4.75E+04**	**6.78E+05**	**3.89E+06**	**4.71E+04**	**1.38E+06**	**3.88E+06**	**4.70E+04**	**1.96E+06**	**3.90E+06**	**4.89E+04**
C_{PM}	5.12E+04	7.14E+05	3.97E+06	5.16E+04	1.46E+06	3.94E+06	5.28E+04	2.02E+06	3.97E+06	5.16E+04
t_{PTS}	**5.10**	19.11	32.53	**4.05**	10.27	12.27	**7.40**	**16.15**	**15.81**	**3.51(64)**
t_{PM}	18.43	7.89	6.81	7.65	5.27	4.16	29.60	29.76	24.65	7.65(128)
23MILLIONS										
C_{PTS}	**9.10E+04**	**9.45E+05**	**5.19E+06**	**9.26E+04**	**1.88E+06**	**5.18E+06**	**9.07E+04**	**2.65E+06**	**5.20E+06**	**9.26E04**
C_{PM}	9.93E+04	9.80E+05	5.36E+06	9.43E+04	1.98E+06	5.33E+06	1.06E+05	2.79E+06	5.37E+06	9.79E+04
t_{PTS}	**12.49**	46.89	74.29	**6.15**	21.50	26.64	**10.25**	20.21	20.67	**6.15(128)**
t_{PM}	43.07	19.67	18.00	12.99	7.36	6.25	17.53	14.00	16.09	10.35(192)
45MILLIONS										
C_{PTS}	**1.15E+05**	**1.13E+06**	**7.24E+06**	**1.11E+05**	**2.52E+06**	**7.26E+06**	**1.06E+05**	**3.65E+06**	**7.29E+06**	**1.05E+05**
C_{PM}	1.26E+05	1.38E+06	7.57E+06	1.33E+05	2.72E+06	7.58E+06	1.39E+05	3.81E+06	7.62E+06	1.26E+05
t_{PTS}	**24.24**	102.29	150.56	**10.69**	39.91	49.51	**13.85**	28.08	30.04	**10.26(192)**
t_{PM}	84.55	48.24	36.21	26.28	17.22	12.55	28.72	25.65	23.15	21.51(256)
82MILLIONS										
C_{PTS}	**1.46E+05**	**1.90E+06**	**1.08E+07**	**1.44E+05**	**3.95E+06**	**1.09E+07**	**1.40E+05**	**5.57E+06**	**1.09E+07**	**1.45E+05**
C_{PM}	1.78E+05	2.12E+06	1.13E+07	1.69E+05	4.18E+06	1.14E+07	1.73E+05	5.95E+06	1.14E+07	1.61E+05
t_{PTS}	**46.48**	189.42	297.76	**17.98**	75.86	91.52	**23.26**	46.91	61.54	**16.93(192)**
t_{PM}	176.40	85.87	76.42	48.38	24.43	21.63	32.83	30.22	26.90	30.00(256)

Table 4.9: Comparison between PT-Scotch (PTS) and ParMeTiS (PM) for representative numbers of processes and parts. C_{PTS} and C_{PM} (t_{PTS} and t_{PM}) are the size of the edge cut (the execution time in seconds) for PTS and PM, respectively. P_{peak} is the number of processors on which PTS and PM recorded peak performance for each graph. The number in parentheses right after the execution time indicates actual P_{peak}. Daggers indicate abortion due to an invalid MPI operation. Data extracted from [58].

Test case	Number of processors:Number of parts									
	32:2	32:32	32:1024	128:2	128:128	128:1024	384:2	384:256	384:1024	P_{peak}:2
AUDIKW1										
C_{PTS}	**1.08E+05**	2.08E+06	1.00E+07	**1.06E+05**	4.22E+06	9.99E+06	**1.05E+05**	5.81E+06	9.96E+06	**1.11E+05**
C_{PM}	1.14E+05	2.04E+06	9.76E+06	1.12E+05	4.15E+06	9.75E+06	1.15E+05	5.76E+06	9.76E+06	1.12E+05
t_{PTS}	**3.51**	11.84	17.35	2.90	8.72	9.29	5.87	10.72	10.06	3.01(128)
t_{PM}	3.90	3.59	5.27	2.42	2.01	2.97	4.45	4.62	4.51	2.37(192)
BRGM										
C_{PTS}	3.46E+05	3.50E+06	2.17E+07	3.49E+05	7.60E+06	2.17E+07	3.30E+05	1.09E+07	2.15E+07	3.49E+05
C_{PM}	†	†	†	†	†	†	†	†	†	†
t_{PTS}	7.86	22.15	58.17	5.34	18.93	24.28	8.39	19.33	19.40	5.34(128)
t_{PM}	†	†	†	†	†	†	†	†	†	†
CAGE15										
C_{PTS}	**7.66E+05**	**3.39E+06**	**9.26E+06**	**7.53E+05**	**5.16E+06**	**8.93E+06**	**7.80E+05**	**6.22E+06**	**8.72E+06**	**7.53E+05**
C_{PM}	8.39E+05	3.98E+06	1.04E+07	8.44E+05	6.05E+06	1.06E+07	7.85E+05	7.32E+06	1.09E+07	8.23E+05
t_{PTS}	31.07	82.96	100.97	29.70	62.80	64.90	41.86	85.30	79.14	29.70(128)
t_{PM}	11.24	9.67	13.13	6.81	5.69	8.90	26.51	25.67	21.42	6.45(64)
COUPOLE8000										
C_{PTS}	**3.08E+03**	**9.56E+04**	**3.17E+06**	**3.08E+03**	**3.92E+05**	**3.17E+06**	**3.08E+03**	**7.88E+05**	**3.17E+06**	**3.08E+03**
C_{PM}	3.13E+03	9.91E+04	3.28E+06	3.20E+03	4.19E+05	3.28E+06	3.14E+03	8.39E+05	3.28E+06	3.14E+03
t_{PTS}	**1.68**	6.76	10.65	**0.83**	2.96	3.73	2.05	4.80	4.88	**0.83(128)**
t_{PM}	3.46	2.84	2.51	1.47	1.62	1.31	0.87	0.89	0.94	0.87(384)
THREAD										
C_{PTS}	**5.60E+04**	6.15E+05	**1.82E+06**	**5.60E+04**	1.03E+06	**1.82E+06**	**5.60E+04**	**1.29E+06**	**1.82E+06**	**5.62E+04**
C_{PM}	5.62E+04	6.03E+05	1.84E+06	5.67E+04	1.02E+06	1.85E+06	5.73E+04	1.29E+06	1.84E+06	5.63E+04
t_{PTS}	**0.53**	0.97	**1.07**	**0.60**	1.08	**1.05**	**0.85**	1.27	**1.28**	**0.47(16)**
t_{PM}	0.77	0.75	1.99	0.70	0.67	1.98	2.00	0.89	2.07	0.52(8)

Table 4.10: Continuation of Table 4.9.

Table 4.9 presents the execution times and the cut sizes yielded by PT-Scotch and ParMeTiS for our test graphs, for representative numbers of processes and parts.

On most of these test cases, the partitions computed by PT-Scotch compare favorably to those produced by ParMeTiS. This gain can be as high as 20 % when bipartitioning graph 82millions, irrespective of the number of processes. PT-Scotch always computes better results for small numbers of parts, while ParMeTiS produces marginally better cuts for three graphs, namely audikw1, thread and brgm, when the number of parts increases. This phenomenon is due to the fact that, to date, PT-Scotch performs k-way partitioning by means of recursive bipartitioning, while ParMeTiS uses a direct k-way algorithm. Consequently, the quality of the partitions produced by PT-Scotch is likely to degrade for large numbers of parts, because of the greedy nature of the recursive bipartitioning scheme which prevents reconsidering earlier choices. It is therefore not surprising that the three graphs for which ParMeTiS gains over PT-Scotch are those of higher degree, for which bad decisions in the earlier bipartitioning stages produce higher penalties in terms of cut.

However, this phenomenon is most often compensated by the improvement in quality yielded by folding-with-duplication and multi-sequential phases (see Section 3.5). Indeed, for most of the test graphs, both PT-Scotch and ParMeTiS exhibit stable partition quality for all numbers of parts and up to 384 processes (which is the largest number of processes in our experiment), as can be seen for instance for graph 10millions in Figure 4.14.

Yet, in the case of graph cage15, partition quality for ParMeTiS decreases as the number of processes and parts increases, while it improves for PT-Scotch. This graph, which is not a mesh, has many topological irregularities (in terms of degrees and connectivity) which are likely to create coarsening artefacts, and therefore require efficient local optimization during the uncoarsening phase of the multilevel framework. As we have exposed in Section 4.1.1, and as had been already evidenced in the context of parallel ordering [24], the relaxation of the sequentiality constraint in the FM implementation of ParMeTiS hinders its efficiency when the number of processes increases.

To compare the relative efficiency of partition quality between PT-Scotch and ParMeTiS based on the data collected, we have plotted the ratio between the cut sizes yielded by these two tools, in Figures 4.15 and 4.16. On mesh graphs, the relative efficiency becomes close to 1 as the number of parts increases. Our assumption is that the decrease in partition quality due to the greedy nature of our recursive bipartitioning algorithm starts to overwhelm the gain of our refinement methods for thousands of parts.

The negative impact of recursive bipartitioning is of course even more perceptible for run time. While PT-Scotch can be more than three times faster than ParMeTiS in the bipartitioning case, such as for graph 82millions, when the number of parts increases, its execution time suffers a penalty factor which tends to a constant proportional to the inverse of the coarsening ratio, as evidenced in Figures 4.17 and 4.18. These figures represent the running times of PT-Scotch for our test graphs, with respect to the numbers of processes and parts. As the number of parts increase, the height of the plots increases and tends to a limit value.

Figure 4.17 plots the execution times of PT-Scotch for graphs 10millions, 23millions, 45millions and 82millions, which have similar topological characteristics. We could not collect data on 2 processors for 45millions, and from 2 to 16 processors for 82millions, as the pieces of the distributed graphs could not fit in the memory of the nodes. The same also happened to ParMeTiS.

In order to analyze the time scalability of PT-Scotch, we focus on the bipartitioning case, for which the penalty factor has no impact. A quick look at the plots shows that PT-Scotch

is scalable up to 64 processors for graph 10millions, and up to 128 processors for 23millions. However, the peak speed is still reached at 128 processors for 45millions, without significant speed-up for 192 processors. For 82millions, we can see a slight speed-up for 192 processors, while the rise in run time which appeared on 256 processors for graph 45millions is reduced. As argued in [71], significant increase in the graph size was required, for ParMeTiS to achieve constant parallel efficiency. This does not seem to be different with PT-Scotch from the asymptotic point of view. Consequently, to evidence some speed-up on 256 processes, a graph at least four times larger than 23millions is required, as the plots suggest.

Figure 4.18 plots the execution times of PT-Scotch for graphs audikw1, coupole8000, cage15, and brgm. Even though these graphs may have different topological properties than those of Figure 4.17, the same scalability properties and limits can be evidenced: the small sizes of the graphs limit the time scalability to 128 processes.

As a last comparison between PT-Scotch and ParMeTiS, the last column of Table 4.9 presents the best execution time, and the associated cut size, obtained by PT-Scotch and ParMeTiS when bipartitioning each of the graphs. We can see that PT-Scotch mostly produces partitions of better quality in smaller time. Moreover, PT-Scotch shows peak performance using less numbers of processors than ParMeTiS for mesh graphs. This lack of scalability is, in our opinion, caused by our recursive bipartitioning scheme, which requires many data movements between processes, all the more when graph pieces are spread across many of them.

4.5.4 Extension to k parts

Considering the advantages of the diffusion scheme in terms of quality and scalability, its extension to k-way partitioning is under way. We assume the behavior of this version should not differ much from the bipartitioning version, as the majority of interface vertices have only two neighbors (see Figure 2.6, page 13). It will be based on the structure of the k-way distributed band graph presented in Figure 4.6 to flow k sorts of liquids from the existing anchor vertices.

In the mean time, basing on a k-way formulation of our band graphs, Meyerhenke *et al.* [84] have already devised a parallel version of their bubble-growing algorithm with a much faster convergence speed because of this reduced problem space. Their implementation provides very good results in term of quality, but the time they take is still long, so we expect that our solution will bring an improvement in this respect.

4.5.5 The barrier of synchronicity

Performing a halo exchange at each iteration, the above parallel formulation of our diffusion algorithm is highly synchronous. With the advent of machines having several hundred thousands of processing elements, and in spite of the continuous improvement of communication subsystems, the demand for more asynchronicity in parallel algorithms is likely to increase. Yet, devising an asynchronous version of the jug of the Danaides algorithm is not so easy as for the original diffusion algorithms. The latter is based on solving a linear system, which may be performed by means of iterative algorithms for which there exist asynchronous formulations [112, 115], while our requirement to remove one unit of liquid per time step creates a threshold effect which cannot be rolled back so easily; it would require every vertex to record the amount of liquid that it could have leaked at each step and did not because there was not enough, so that late incoming quantities can be cut by the amount that would have leaked in the past rounds.

In this respect, GAs may offer a better alternative, as computations can take place asynchronously within independent demes. When enough of them exist, convergence is likely to be

achieved even if some of them may lag behind.

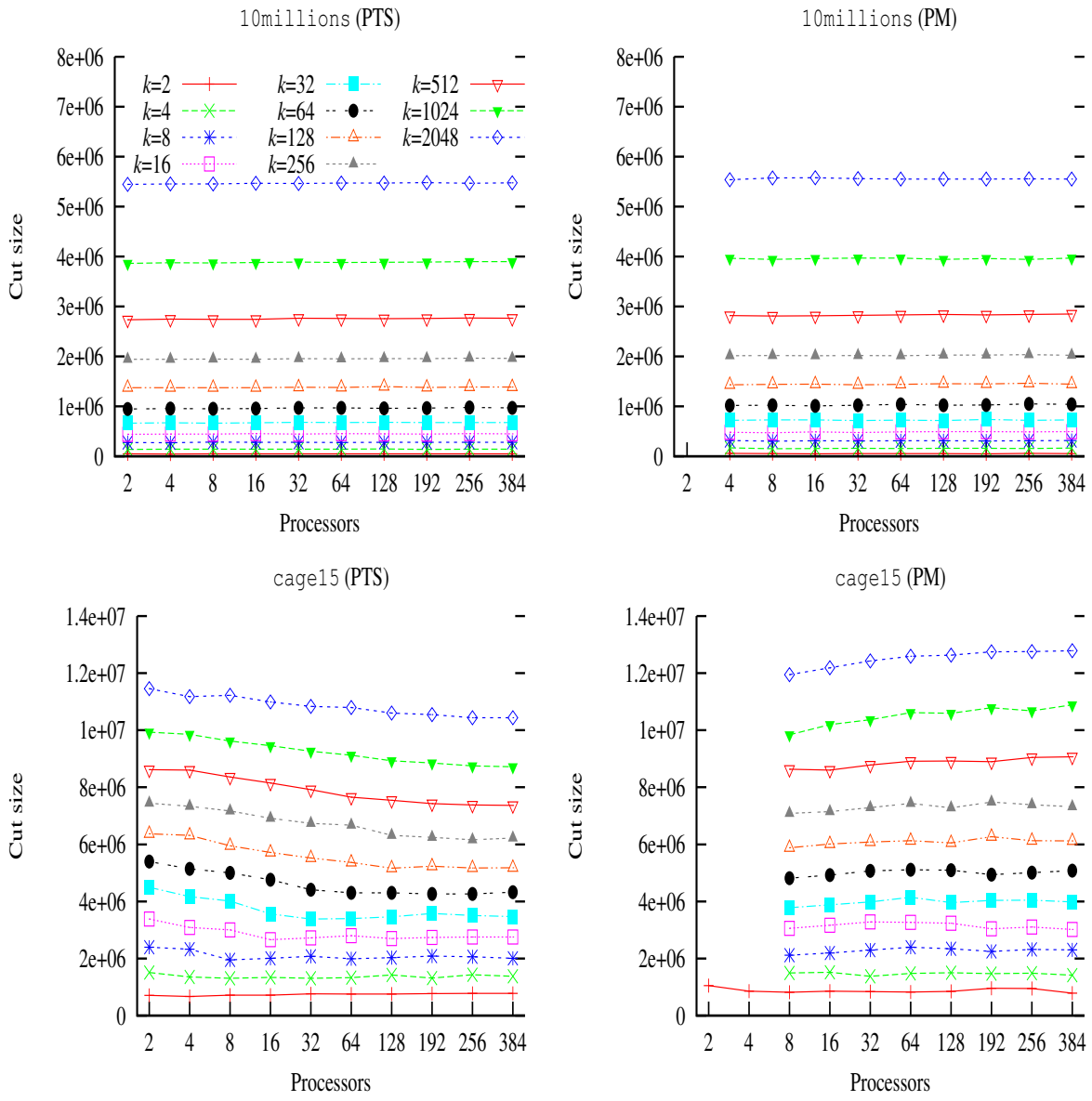


Figure 4.14: Cut sizes of PT-Scotch (PTS, left) and ParMeTiS (PM, right) for graphs 10millions and cage15. Data extracted from [58].

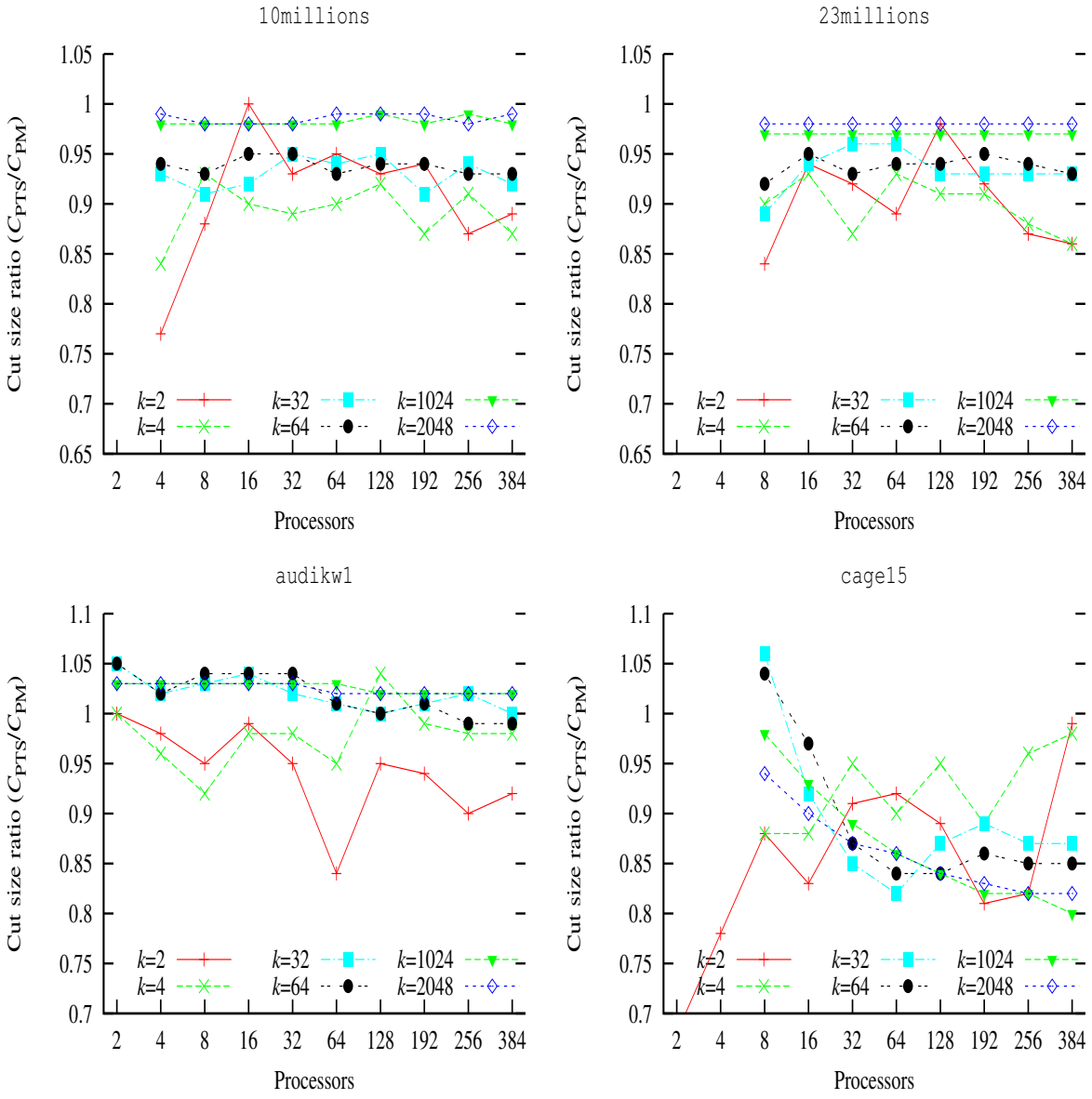


Figure 4.15: Cut size ratio between PT-Scotch (PTS) and ParMeTiS (PM) for graphs 10millions, 23millions, audikw1, and cage15. C_{PTS} and C_{PM} are the size of the edge cut for PTS and PM, respectively. Data extracted from [58].

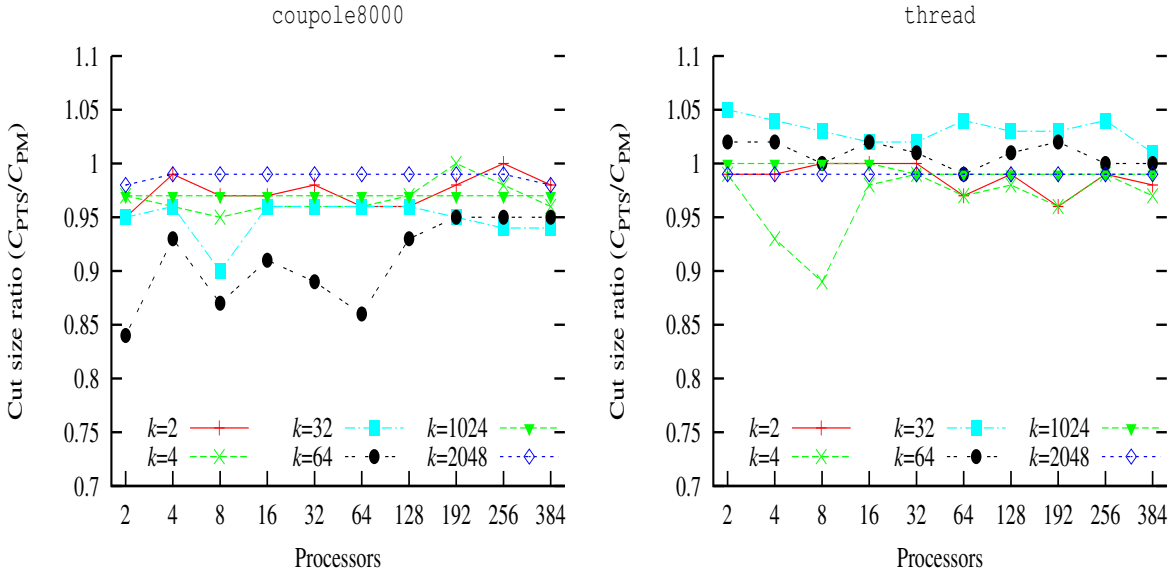


Figure 4.16: Cut size ratio between PT-Scotch (PTS) and ParMeTiS (PM) for graphs coupole8000 and thread. C_{PTS} and C_{PM} are the size of the edge cut for PTS and PM, respectively. Data extracted from [58].

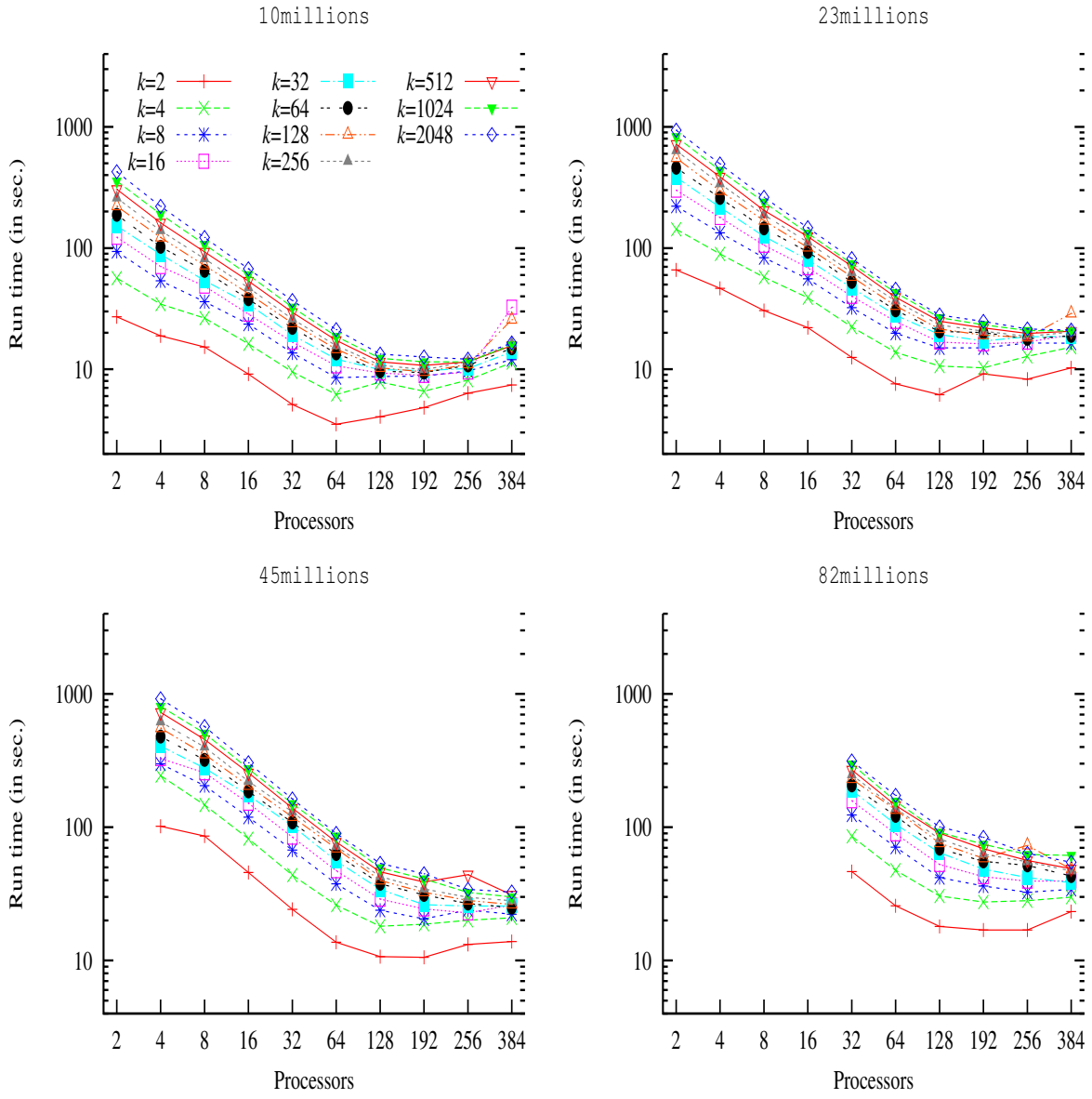


Figure 4.17: Execution times of PT-Scotch for graphs 10millions, 23millions, 45millions and 82millions. Data extracted from [58].

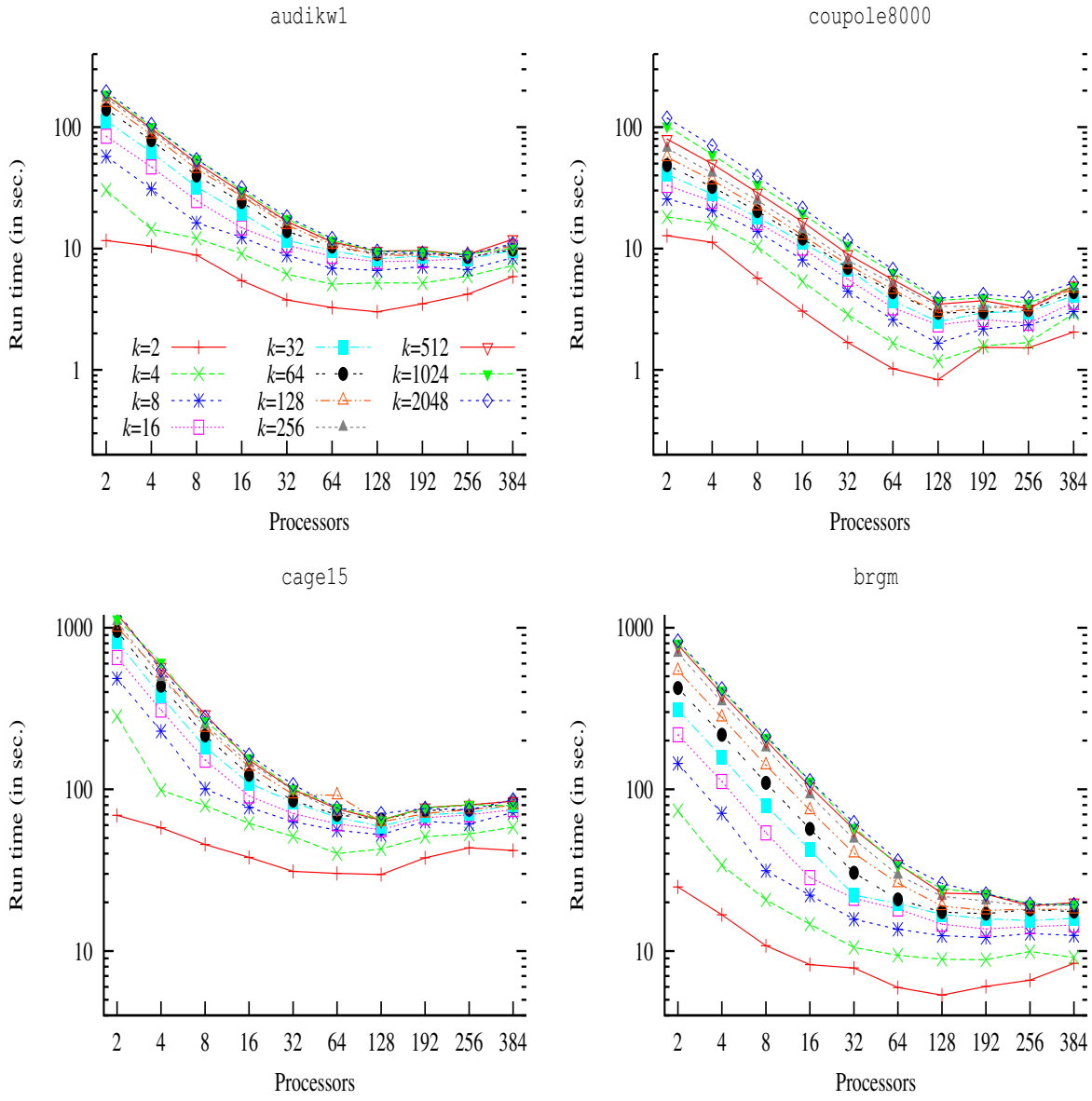


Figure 4.18: Execution times of PT-Scotch for graphs audikw1, coupole8000, cage15 and brgm. Data extracted from [58].

Chapter 5

Conclusion and future works

5.1 Where we are now

As the final results of last chapter show, most of the goals of our initial roadmap are about to be fulfilled by the time this dissertation is written. While we have been able to run PT-Scotch in parallel on more than one thousand processing elements, we lack real-world test graphs of a billion vertices to experiment with. In this respect, some work has to be done regarding I/O: even for graphs as small as 82millions, we are experiencing bottlenecks problem as graph load time is dominating computation time.

To date, PT-Scotch can only compute graph partitions in parallel by means of recursive bipartitioning, but thanks to the upcoming parallel k-way partitioning scheme based on distributed k-way band graphs, we are also about to release a new version providing parallel static mapping features, which are likely to gain popularity as large parallel systems are more and more heterogeneous.

The Scotch software package currently comprises a bit less than one hundred thousand lines of C code. It is distributed at no charge, under the CeCILL-C free software license, to all interested parties, whether they are simple users, researchers working on the same subjects, or potential contributors. We plan to improve Scotch regularly, according to the new developments of our research, so that we remain under the scrutiny of peer evaluation.

5.2 Where we are heading to

In spite of the results already achieved, much is still to be done. We present in the following paragraphs some directions and medium-term goals for future research, which have already started.

5.2.1 Algorithmic issues

First, the scalability of our existing algorithms has to be thoroughly analyzed in the context of large-scale and potentially heterogeneous systems. We do not plan to have PT-Scotch ported on heterogeneous processor architectures such as the Cell [5], because data locality of irregular graph algorithms does not seem sufficient to take advantage of them. However, we have to investigate how the interconnection networks of such large heterogeneous machines can sustain the load of inter-processor collective communication. As suggested in the previous chapter, more room may have to be made for asynchronous algorithms if collective communication latency becomes a problem.

5.2.2 Dynamic graph repartitioning

Second, since I/O is a bottleneck and all data cannot be repartitioned from scratch, it is essential to provide dynamic repartitioning capabilities in PT-Scotch. Many methods already exist in the literature, as well as many existing tools. We plan to take advantage of the static mapping capabilities of Scotch to coerce new mappings to be as close as possible to the original ones. This graph-based formulation will have to be compared to diffusion-based, vertex migration methods.

5.2.3 Adaptive dynamic mesh partitioning

Many simulations which model the evolution of a given phenomenon along with time (turbulence and unsteady flows, for instance) need to re-mesh some portions of the problem graph in order to capture more accurately the properties of the phenomenon in areas of interest. This re-meshing is performed according to criteria which are closely linked to the undergoing computation and can involve large mesh modifications: while elements are created in critical areas, some may be merged in areas where the phenomenon is no longer critical.

Performing such remeshing in parallel creates additional problems. In particular, splitting an element which is located on the frontier between several processors is not an easy task, because deciding when splitting some element, and defining the direction along which to split it so as to preserve numerical stability most, requires shared knowledge which is not available in distributed memory architectures. Ad-hoc data structures and algorithms have to be devised so as to achieve these goals without resorting to extra communication and synchronization which would impact the running speed of the simulation.

Basing on a sequential (re)mesher such as MMG [32], as well as on the dynamic graph repartitioning routines provided by PT-Scotch, we aim to create a parallel framework for dynamic mesh adaption, remeshing and load balancing.

As former INRIA CEO Gilles Kahn said in his last video conference to personnel: "*One always overestimates the work (s)he is able to do in one year, but one always underestimates the results that (s)he is able to achieve in five years*".

I hope the five years to come will be as interesting as the previous five.

Appendix A

Bibliography

[1] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. and Appl.*, 17:886–905, 1996.

[2] P. Amestoy, I. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184:501–520, 2000.

[3] F. André and J.-L. Pazat. Le placement de tâches sur des architectures parallèles. *Technique et Science Informatiques*, pages 385–401, April 1988.

[4] S. Areibi and Y. Zeng. Effective memetic algorithms for VLSI design automation = genetic algorithms + local search + multi-level clustering. *Evolutionary Computation*, 12(3):327–353, 2004.

[5] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond. *Programming the Cell Broadband Engine^{TM} Architecture – Examples and Best Practices.* Redbooks. IBM, 2008.

[6] C. Ashcraft, S. Eisenstat, J. W.-H. Liu, and A. Sherman. A comparison of three column based distributed sparse factorization schemes. In *Proc. Fifth SIAM Conf. on Parallel Processing for Scientific Computing*, 1991.

[7] S. T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proc. ACM/IEEE Conference on Supercomputing (CDROM)*, December 1995.

[8] 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.

[9] R. Battiti and A. A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Trans. Comput.*, 48(4):361–385, 1999.

[10] C. Berge. *Graphs and Hypergraphs.* North Holland Publishing, 1973.

[11] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computing*, C-30(3):207–214, 1981.

[12] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *Proc. 11th Int. Conf. on Parallel Processing*, pages 1–7. The Penn. State Univ. Press, August 1988.

[13] T. N. Bui and B. R. Moon. Genetic algorithm and graph partitioning. *IEEE Trans. Comput.*, 45(7):841–855, 1996.

[14] U. Çatalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Par. and Dist. Syst.*, 10(7):673–693, 1999.

[15] U. Çatalyurek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proc. ACM/IEEE conference on Supercomputing (CDROM)*, 2001. 28 pages.

[16] CeCILL. "CEA-CNRS-INRIA Logiciel Libre" free/libre software license. Available from `http://www.cecill.info/licenses.en.html`.

[17] A. Chan, P. Dehne, F. Bose, and M. Latzel. Coarse grained parallel algorithms for graph matching. *Parallel Computing*, 34(1):47–62, 2008.

[18] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55:463–476, 1989.

[19] S. Chaumette. A software environment for programming distributed memory machines. In *Proc. 25th Hawaii International Conference on System Sciences*, volume 1, pages 257–266. IEEE, 1992.

[20] S. Chaumette and M.-C. Counilh. A development environment for distributed systems. In *Proc. European Distributed Memory Computing Conference, Minehen*, volume 487 of *LNCS*, pages 110–119, 1991.

[21] T.-Y. Chen, J. R. Gilbert, and S. Toledo. Toward an efficient column minimum degree code for symmetric multiprocessors. In *Proc. 9th SIAM Conf. on Parallel Processing for Scientific Computing, San-Antonio*, 1999.

[22] C. Chevalier. *Conception et mise en œuvre d'outils efficaces pour le partitionnement et la distribution parallèles de problèmes numériques de très grande taille.* Thèse de Doctorat, LaBRI, Université Bordeaux I, September 2007.

[23] C. Chevalier and F. Pellegrini. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *Proc. Euro-Par'06, Dresden*, volume 4128 of *LNCS*, pages 243–252, September 2006.

[24] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.

[25] C. Chevalier and I. Safro. Comparison of coarsening schemes for multilevel graph partitioning. In *Proc. LION'3 Combinatorial Scientific Computing*, Trento, Italy, January 2009.

[26] C. Chevalier and I. Safro. Weighted aggregation for multi-level graph partitioning. In *Combinatorial Scientific Computing*, number 09061 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.

[27] M.-C. Counilh and J. Roman. Expression for massively parallel algorithms – description and illustrative example. *Parallel Computing*, 16(2–3):239–251, 1990.

[28] T. Davis. University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[29] R. Diekmann, D. Meyer, and B. Monien. Parallel decomposition of unstructured FEM-meshes. *Concurrency: Practice & Experience*, 10(1):53–72, 1998.

[30] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. In *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–73, 1995.

[31] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Aspect ratio for mesh partitioning. In *Proc. Euro-Par'98*, volume 1470 of *LNCS*, pages 347–351, 1998.

[32] C. Dobrzynski. 3d local anisotropic remeshing for rigid bodies displacement. In *Second Workshop on Grid Generation for Numerical Computations , INRIA Rocquencourt*, October 2007. `http://www-rocq.inria.fr/who/Frederic.Alauzet/tetra\_eng.html`.

[33] I. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Soft.*, 7(3):315–330, September 1981.

[34] 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.

[35] C. Fahrat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[36] M. Faverge, X. Lacoste, and P. Ramet. A NUMA aware scheduler for a parallel sparse direct solver. In *Proc. PMAA*, June 2008.

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

[38] Jérémie Gaidamour and Pascal Hénon. A parallel direct/iterative solver based on a Schur complement approach. In *Proc. 11^{th} Int. Conf. on Comp. Sci. and Eng.*, pages 98–105, Sao Paulo, 2008.

[39] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

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

[41] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[42] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

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

[44] A. George and J. W.-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[45] GNU. Lesser General Public License. Available from: `http://www.gnu.org/copyleft/lesser.html`.

[46] G. H. Golub and C. F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[47] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman, and J.-J. Pesqué. Description of the Emilio software processing chain and application to structural mechanics. In *Proceedings of PMAA*, August 2000.

[48] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman, and J.-J. Pesqué. Parallel sparse linear algebra and application to structural mechanics. *Numerical Algorithms*, 24:371–391, 2000.

[49] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel Distrib. Syst.*, 8(5):502–520, 1997.

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

[51] B. Hendrickson. Combinatorial scientific computing. `http://www.sandia.gov/~bahendr/csc.html`.

[52] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *IRREGULAR'98: solving irregularly structured problems in parallel*, number 1457 in LNCS, pages 218–225, August 1998.

[53] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. ACM/IEEE conference on Supercomputing (CDROM)*, December 1995. 28 pages.

[54] B. Hendrickson, R. Leland, and R. Van Driessche. Skewed graph partitioning. In *Proceedings of the 8^{th} SIAM Conference on Parallel Processing for Scientific Computing*. IEEE, March 1997.

[55] B. Hendrickson and A. Pothen. Combinatorial scientific computing: The enabling power of discrete algorithms in computational science. In *High Performance Computing for Computational Science - VECPAR 2006*, number 4395 in LNCS, pages 260–280, 2007.

[56] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, 1998.

[57] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, January 2002.

[58] J.-H. Her and F. Pellegrini. Efficient and scalable parallel graph partitioning by recursive bipartitioning. *Parallel Computing*, 2010. To appear. `http://www.labri.fr/~pelegrin/papers/scotch_parallelbipart_parcomp.pdf`.

[59] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Harbor, 1975.

[60] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, December 1973.

[61] J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched Pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pages 82–87, 1994.

[62] JOSTLE: Graph partitioning software. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[63] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proc. ACM/IEEE conference on Supercomputing (CDROM)*, December 1995. 19 pages.

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

[65] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press, 1995.

[66] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In *Proc. 8^{th} SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[67] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[68] G. Karypis and V. Kumar. METIS- *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minnesota, September 1998.

[69] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[70] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[71] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[72] A. Kaveh and H. A. B. Rahimi. A hybrid graph-genetic method for domain decomposition. *Finite. Elem. Anal. Des.*, 29:1237–1247, 2003.

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

[74] A. I. Khan and B. H. V. Topping. Subdomain generation for parallel finite element analysis. *Comput. Syst. Eng.*, 4:96–129, 1998.

[75] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[76] M. Laguna, T. A. Feo, and H. C. Elrod. A greedy randomized adaptative search procedure for the two-partition problem. *Operations Research*, pages 677–687, August 1994.

[77] A. E. Langham and P. W. Grant. Using competing ant colonies to solve k-way partitioning problems with foraging and raiding strategies. In *ECAL'99: Proc. 5^{th} European Conference on Advances in Artificial Life*, number 1674 in LNCS, pages 621–625, 1999.

[78] C. Leiserson and J. Lewis. Orderings for parallel sparse symmetric factorization. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987.

[79] J.-M. Lépine and F. Rubi. The CHEOPS operating system. In *Proc. First European Workshop on Hypercube and Distributed Computers*, pages 161–174. North-Holland, 1989.

[80] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numerical Analysis*, 16(2):346–358, April 1979.

[81] J. W.-H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Soft.*, 11(2):141–153, 1985.

[82] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, 1986.

[83] MeTiS: Family of multilevel partitioning algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[84] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *Proc. 22nd IPDPS*, pages 1–13, 2008.

[85] H. Meyerhenke and S. Schamberger. Balancing parallel adaptive FEM computations by solving systems of linear equations. In *Proc. Euro-Par'05*, volume 3648 of *LNCS*, pages 209–219, 2005.

[86] H. Meyerhenke and S. Schamberger. A parallel shape optimizing load balancer. In *Proc. Euro-Par'06*, volume 4128 of *LNCS*, pages 232–242, 2006.

[87] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[88] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts, towards memetic algorithms. Technical Report 826, California Intitute of Technology, 1989.

[89] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In A. K. Noor, editor, *Parallel Computations and Their Impact on Mechanics*, pages 209–227. ASME Press, 1986.

[90] T. Oakenshield. *Recursive Bipartitioning of Trolls with a Two-Handed Axe*. Moria Press, 1947.

[91] PARASOL project (EU ESPRIT IV LTR Project No. 20160), 1996–1999.

[92] J.-L. Pazat and B. Vauquelin. MAPP: un mécanisme d'aide au placement des processus sur multiprocesseur. In *Actes des journées du pôle architecture C^3*, volume 56 of *Bigre + Globule*, pages 116–124, 1987.

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

[94] F. Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*. Thèse de Doctorat, LaBRI, Université Bordeaux I, January 1995.

[95] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proc. Euro-Par'07*, volume 4641 of *LNCS*, pages 191–200. Springer, August 2007.

[96] F. Pellegrini. SCOTCH *and* LIBSCOTCH 5.1 *User's Guide*. LaBRI, Université Bordeaux I, August 2008. Available from `http://www.labri.fr/~pelegrin/scotch/`.

[97] F. Pellegrini. Distillating knowledge about Scotch. In *Combinatorial Scientific Computing*, number 09061 in Dagstuhl Seminar Proceedings, February 2009. 12 pages.

[98] F. Pellegrini and D. Goudin. Using the native mesh partitioning capabilities of Scotch 4.0 in a parallel industrial electromagnetics code. In *Eleventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, USA*, February 2004.

[99] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. HPCN'96, Brussels*, volume 1067 of *LNCS*, pages 493–498, April 1996.

[100] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proc. HPCN'97*, volume 1225 of *LNCS*, pages 370–378, April 1997.

[101] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000.

[102] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Software*, 16(4):303–324, December 1990.

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

[104] A. Rama Mohan Rao. Distributed evolutionary multi-objective mesh-partitioning algorithm for parallel finite element computations. *Computers & Structures*, 87, number =.

[105] A. Rama Mohan Rao, T. V. S. R. Appa Rao, and B. Dattaguru. Automatic decomposition of unstructured meshes employing genetic algorithms for parallel FEM computations. *Int. J. Struct. Eng. Mech.*, 14:625–647, 2002.

[106] SCALAPPLIX: Algorithms and high performance computing for grand challenge applications. `http://www.inria.fr/recherche/equipes/scalapplix.en.html`.

[107] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

[108] R. Schreiber. Scalability of sparse direct solvers. Technical Report TR 92.13, RIACS, NASA Ames Research Center, May 1992.

[109] SCOTCH: Static mapping, graph partitioning, and sparse matrix block ordering package. `http://www.labri.fr/~pelegrin/scotch/`.

[110] H. D. Simon and S.-H. Teng. How good is recursive bisection. *SIAM J. Sci. Comput.*, 18(5):1436–1445, September 1997.

[111] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *J. of Global Optimization*, 29(2):225–241, 2004.

[112] D. B. Szyld. Different models of parallel asynchronous iterations with overlapping blocks. *Computational and Applied Mathematics*, 17:101–115, 1998.

[113] E.-G. Talbi and P. Bessière. A parallel genetic algorithm for the graph partitioning problem. In *ICS'91: Proceedings of the 5^{th} international conference on Supercomputing*, pages 312–320. ACM, 1991.

[114] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, 1967.

[115] G. Tongxiang. Asynchronous relaxed iterative methods for solving linear systems of equations. *Applied Mathematics and Mechanics*, 18(8):801–806, August 1997.

[116] R. Vanderstraeten, R. Keunings, and C. Farhat. Beyond conventional mesh partitioning algorithms. In *SIAM Conf. on Par. Proc.*, pages 611–614, 1995.

[117] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[118] C. Walshaw and M. Cross. Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000.

[119] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[120] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comput. Syst.*, 17(5):601–623, 2001.

[121] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

[122] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for unstructured meshes. Technical Report 97/IM/20, University of Greenwich, London SE18 6PF, UK, March 1997.

[123] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In *Proc. Irregular'95*, number 980 in LNCS, pages 121–126, 1995.

[124] Y. Wan, S. Roy, A. Saberi, and B. Lesieutre. A stochastic automaton-based algorithm for flexible and distributed network partitioning. In *Proc. Swarm Intelligence Symposium*, pages 273–280. IEEE, 2005.

[125] D. Whitley. A genetic algorithm tutorial. *Mathematics and Statistics*, 4:65–85, June 1994.

[126] D. Whitley, S. Rana, and R. B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–47, 1999.

Contributions au partitionnement de graphes parallèle multi-niveaux

Le partitionnement de graphes est une technique employée dans de nombreux domaines scientifiques. Il est utilisé pour résoudre des problèmes d'optimisation, modélisés sous la forme de graphes valués ou non, et pour lesquels la recherche de bonnes solutions équivaut au calcul, éventuellement récursivement, de coupes sommet ou arête les plus petites possible et qui équilibrent les tailles des sous-parties séparées. La plupart des méthodes actuelles de partitionnement de graphes mettent en œuvre un schéma multi-niveaux, dans lequel le graphe à partitionner est successivement contracté pour former une famille de graphes de plus en plus petits, mais de structure topologique similaire, de sorte qu'une partition initiale calculée sur le plus petit graphe puisse être propagée de proche en proche, par prolongations et raffinements successifs, jusqu'à obtenir un partitionnement du graphe initial.

Du fait de l'augmentation croissante de la taille des problèmes à résoudre, ceux-ci ne peuvent plus être traités de façon séquentielle sur un unique ordinateur. Il est donc nécessaire de concevoir des algorithmes parallèles de partitionnement de graphes, aptes à traiter des graphes à plusieurs milliards de sommets distribués sur plusieurs milliers de processeurs. Plusieurs auteurs s'étaient déjà attelés à cette tâche, mais la performance des algorithmes proposés, ou la qualité des solutions produites, se dégradent lorsque le nombre de processeurs augmente.

Ce mémoire présente les travaux réalisées au sein du projet PT-Scotch sur la conception d'algorithmes efficaces et robustes pour la parallélisation du schéma multi-niveaux. Il se concentre en particulier sur les phases de contraction et de raffinement, qui sont les plus critiques en termes de performance et de qualité des solutions produites. Il propose un algorithme parallèle probabiliste d'appariement, ainsi qu'un ensemble de méthodes permettant de réduire l'espace des solutions au cours la phase de raffinement et facilitant l'usage de méthodes globales, qui passent mieux à l'échelle mais sont en général bien plus coûteuses que les algorithmes d'optimisation locale habituellement mis en œuvre dans le cas séquentiel.

Contributions to parallel multilevel graph partitioning

Graph partitioning is a technique which has applications in many fields of science. It is used to solve domain-dependent optimization problems, modeled in terms of weighted or unweighted graphs, where finding good solutions amounts to computing, eventually recursively, smallest possible vertex or edge cuts which balance evenly the weights of the separated parts. Most of current graph partitioning methods implement a multilevel framework, in which the graph to partition is successively coarsened to create a family of smaller graphs, of similar topological structure, so that an initial partition computed on the coarsest graph can be propagated, from coarser to finer graphs, by prolongation and refinement of the prolonged partitions, up to obtain a partition of the original graph.

Because of the ever increasing size of the problems to solve, these can no longer be handled sequentially on a single computer. It is therefore necessary to devise parallel graph partitioning algorithms, suitable for the handling of graphs with several billion vertices distributed over several thousands of processing elements. Several authors already tackled this task, but either the performance of the proposed algorithms, or the quality of the solutions produced, decrease when the number of processing elements increases.

This dissertation presents the works carried out, within the PT-Scotch project, on the design and implementation of efficient and robust algorithms for the parallelization of the multilevel framework. It focuses particularly on the coarsening and refinement phases, which are the most critical in terms of performance and of quality of the produced solutions. It presents a

probabilistic parallel matching algorithm, as well as a set of methods allowing to reduce the the solution space during the refinement phase, allowing for the use of global methods, which are more scalable but are generally much more expensive than the local optimization algorithms which are commonly used in the sequential case.