PT-Scotch: A tool for efficient parallel graph ordering

C. Chevalier[a*], F. Pellegrini[b]

[a]LaBRI & Project ScAlApplix of INRIA Futurs
351, cours de la Libération, 33400 Talence, France

[b]ENSEIRB, LaBRI & Project ScAlApplix of INRIA Futurs
351, cours de la Libération, 33400 Talence, France
{cchevali|pelegrin}@labri.fr

The parallel ordering of large graphs is a difficult problem, because neither minimum-degree algorithms, nor the best graph partitioning methods that are necessary to nested dissection, parallelize or scale well. This paper presents a set of algorithms, implemented in the PT-Scotch software package, which allows one to order large graphs in parallel, yielding orderings the quality of which is equivalent to the one of state-of-the-art sequential algorithms.

1. Introduction

Graph partitioning is an 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, eventually recursively in a divide-and-conquer framework, small vertex or edge cuts that balance evenly the weights of the graph parts.

Because there always exists large problem graphs which cannot fit in the memory of sequential computers and cost too much to partition, parallel graph partitioning tools have been developed [1,2], but their outcome is mixed. In particular, in the context of parallel graph ordering which is the one of this paper, they do not scale well, as partitioning quality tends to decrease, and thus fill-in tends to increase much, when the number of processors which run the program increase.

The purpose of the PT-Scotch software ("*Parallel Threaded* Scotch", an extension of the sequential Scotch software), developed at LaBRI within the ScAlApplix project of INRIA Futurs, is to provide efficient parallel partitioning of graphs with sizes up to a billion vertices, distributed over a thousand processors. Scalability issues have therefore to receive much attention.

One of the target applications of PT-Scotch within the ScAlApplix project is graph ordering, which is a critical problem for the efficient factorization of symmetric sparse matrices, not only to reduce fill-in and factorization cost, but also to increase concurrency in the elimination tree, which is essential in order to achieve high performance when

*This author's work is funded by a joint PhD grant of CNRS and Région Aquitaine.

solving these linear systems on parallel architectures. We therefore focus in this paper on this specific problem, although we expect some of the algorithms presented here to be reused in the near future in the context of edge, k-way partitioning.

The two most classically-used reordering methods are Minimum Degree and Nested Dissection. The Minimum Degree algorithm [3] is a local heuristic which is extremely fast and very often efficient, but it is intrinsically sequential, so that attempts to derive parallel versions of it have not been successful [4], especially for distributed-memory architectures. The Nested Dissection method [5], on the other hand, is very suitable for parallelization, since it consists in computing a small vertex set that separates the graph into two parts, ordering the separator vertices with the highest indices available, then proceeding recursively on the two separated subgraphs until their size is smaller than a specified threshold.

This paper presents the algorithms which have been implemented in PT-Scotch to parallelize the Nested Dissection method. The distributed data structures used by PT-Scotch will be presented in the next section, while the algorithms that operate on them will be described in Section 3. Section 4 will show some results, and the concluding section will be devoted to discussing some on-going and future work.

2. Distributed data structures

Since PT-Scotch extends the graph ordering capabilities of Scotch in the parallel domain, it has been necessary to define parallel data structures to represent distributed graphs as well as distributed orderings.

2.1. Distributed graph

Like for centralized graphs in Scotch as well as in other software packages, distributed graphs are classically represented in PT-Scotch by means of adjacency lists. Vertices are distributed across processors along with their adjacency lists and with some duplicated global data, as illustrated in Figure 1. Global data comprise `baseval`, the starting index of all numberings, which can be set to 0 for C-style arrays or to 1 for Fortran-style arrays, and `procglbnbr`, the number of processors across which the graph is distributed. In order to allow users to create and destroy vertices without needing any global renumbering, every processor is assigned a user-defined range of global vertex indices, recorded in the `proc dsptab` array which is also duplicated. Local subgraphs located on every processor can therefore be updated independently, as long as the number of vertices possessed by some process p does not exceed ($\texttt{procdsptab}[p+1] - \texttt{procdsptab}[p]$).

Since many algorithms require that local data be attached to every vertex, and since global indices cannot be used for that purpose, all vertices owned by any processor p are also assigned local indices, suitable for the indexing of compact local data arrays. These local indices range from `baseval` to ($\texttt{proccnttab}[p] + \texttt{baseval}) - 1$, inclusive, and the corresponding global number of any local vertex index i is therefore ($\texttt{procdsptab}[p]+\mathrm{i}-\texttt{baseval}$). 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 processor number and by ascending global number, such that, when vertex data have to be exchanged between neighboring processors, these data can be agglomerated in a cache-friendly way on the

sending side, 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 beared by local vertices to the ghost copies possessed by all of its neighboring processors. This low-level routine is used by many algorithms of PT-Scotch, for instance to spread vertex labels of selected vertices in the induced subgraph building routine (see Section 3.1), or to share matching data in the coarse graph building routine (see Section 3.2). This routine is also available, as a collective communication routine, to PT-Scotch library users, who have to provide on every processor a reference to the vertex data array they want to diffuse, the size of which should be equal to `vertgstnbr`, the overall number of local and ghost vertices possessed by the processor, and the local entries of which are filled with useful data to spread, along with the MPI datatype of the array cells.

Because global and local indexings coexist, two adjacency arrays are in fact maintained on every processor. The first one, `edgeloctab`, usually provided by the user, holds the global indices of the neighbors of any given vertex, while the second one, `edgegsttab`, which is internally maintained by PT-Scotch, holds the local and ghost indices of the neighbors. The starting index of the adjacency list of some local vertex i in the processor's adjacency array is given by `vertloctab[`i`]`, and its after-end index by `vendloctab[`i`]`. However, when adjacency arrays are fully ordered and without any unused space, such as for subgraphs created by PT-Scotch itself during its multi-level nested dissection ordering process, only one `vertloctab` array, of size (`vertlocnbr` $+ 1$) needs to be allocated, and `vendloctab` is set to point to (`vertloctab` $+ 1$), since `vendloctab`$[i]$ is then always equal to `vertloctab`$[i + 1]$. Since only local vertices are processed by the distributed algorithms, the adjacency of ghost vertices is never stored on the processors.

When the `edgegsttab` arrays are created, some global, duplicated data is aggregated for future use, such as `proccnttab`, an array of size `procglbnbr` which contains the number of local vertices possessed by every processor, and `procvrttab`, which contains the prefix sum, starting from `baseval`, of the entries of `proccnttab`, both of which are used when gathering vertex data by means of the `MPI_Gather` routine.

2.2. Distributed ordering

During its execution, PT-Scotch builds a distributed tree structure, spreading on all of the processors onto which it is run, and the leaves of which represent fragments of the inverse permutation describing the computed ordering. We use the inverse permutation rather than the direct permutation because the inverse permutation can be built and optimized in a fully distributed way: every subgraph to be reordered is described only by the number of vertices to reorder, and by the smallest index of the new ordering to assign to the subgraph vertices. Once a subgraph (either a separator or a leaf of the separation tree) is to be reordered, a new permutation fragment is created on every processor which owns some of its vertices. The size of a fragment is the number of vertices of the subgraph that are locally owned by the processor, and the starting ordering index of the fragment is the smallest ordering index assigned to the subgraph, plus the sum of the sizes of the subgraph permutation fragments owned by processors of smaller ranks. The order in which the global indices of the subgraph vertices are located in the fragment describes how subgraph vertices are ordered in the inverse permutation.

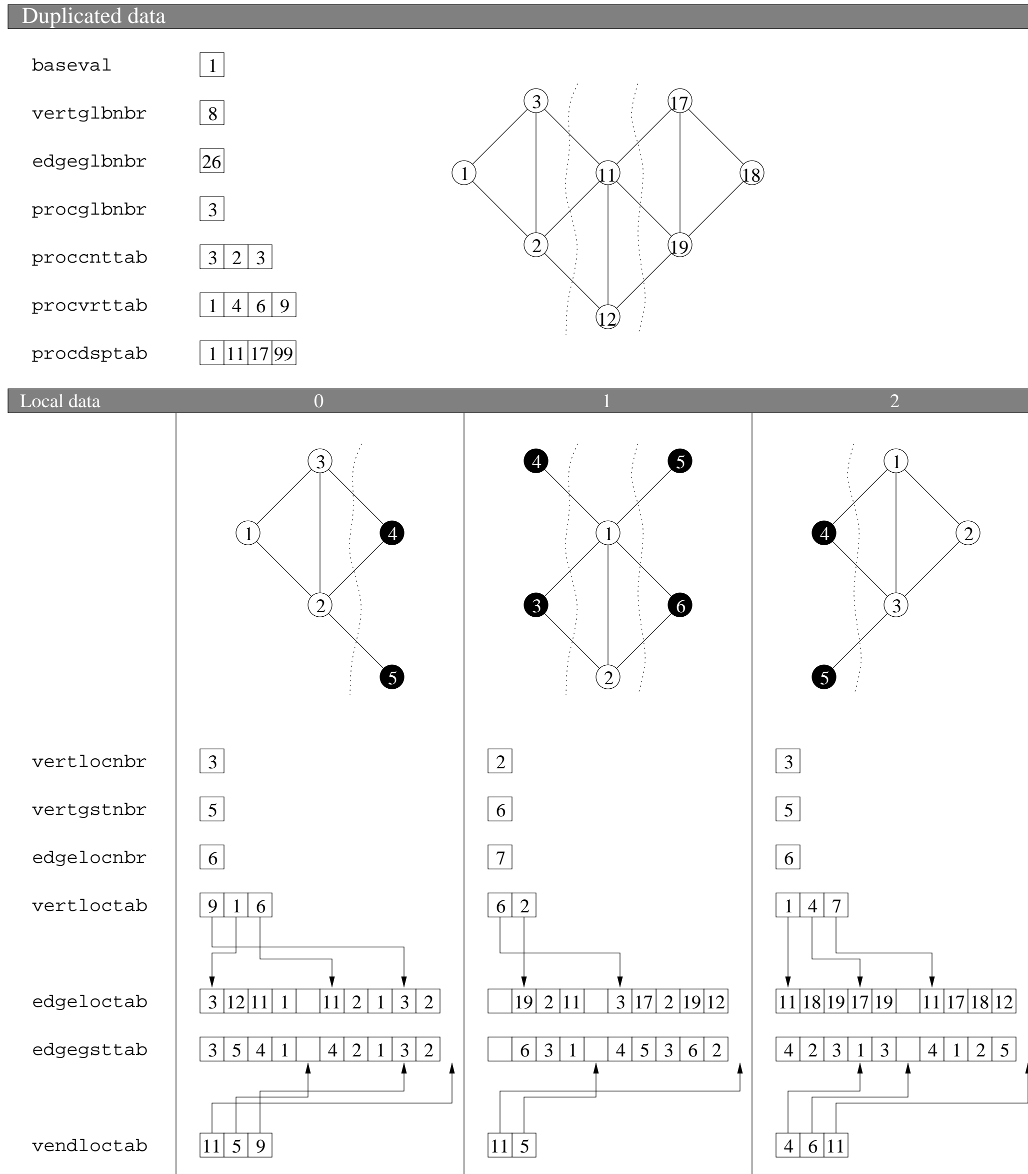

Figure 1. Data structures of a graph distributed across three processors. The global image of the graph is shown above, while the three partial subgraphs owned by the three processors are represented below. Adjacency arrays with global vertex indexes are stored in `edgeloctab` arrays, while local compact numberings of local and ghost neighbor vertices are internally available in `edgegsttab` arrays. Local vertices owned by every processor are drawn in white, while ghost vertices are drawn in black. For each local vertex i located on processor p, the global index of which is ($\texttt{procdsptab}[p] + i - \texttt{baseval}$), the starting index of the adjacency array of i in `edgeloctab` (global indices) or `edgegsttab` (local indices) is given by $\texttt{vertloctab}[i]$, and its after-end index by $\texttt{vendloctab}[i]$. For instance, local vertex 2 on processor 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`.

3. Algorithms for efficient parallel reordering

The parallel computation of orderings in PT-Scotch involves three different levels of concurrency, corresponding to three key steps of the nested dissection process: the nested dissection algorithm itself, the multi-level coarsening algorithm used to compute separators at each step of the nested dissection process, and the refinement of the obtained separators. Each of these steps is described below.

3.1. Nested dissection

As said above, the first level of concurrency relates to the parallelization of the nested dissection method itself, which is straightforward thanks to the intrinsically concurrent nature of the algorithm. Starting from the initial graph, arbitrarily distributed across p processors but preferably balanced in terms of vertices, the algorithm proceeds as illustrated in Figure 2 : once a separator has been computed in parallel, by means of a method described below, each of the p processors participates in the building of the distributed induced subgraph corresponding to the first separated part (even if some processors do not have any vertex of it). This induced subgraph is then folded onto the first $\lceil \frac{p}{2} \rceil$ processors, such that the average number of vertices per processor, which guarantees efficiency as it allows the shadowing of communications by a subsequent amount of computation, remains constant. During the folding process, vertices and adjacency lists owned by the $\lfloor \frac{p}{2} \rfloor$ sender processors are redistributed to the $\lceil \frac{p}{2} \rceil$ receiver processors so as to evenly balance their loads.

The same procedure is used to build, on the $\lfloor \frac{p}{2} \rfloor$ remaining processors, the folded induced subgraph corresponding to the second part. These two constructions being completely independent, the computations of the two induced subgraphs and their folding can be performed in parallel, thanks to the temporary creation of an extra thread per processor. When the vertices of the separated graph are evenly distributed across the processors, this feature favors load balancing in the subgraph building phase, because processors which do not have many vertices of one part will have the rest of their vertices in the other part, thus yielding the same overall workload to create both graphs in the same time. This feature can be disabled when the communication system of the target machine is not thread-safe.

At the end of the folding process, every processor has a folded subgraph fragment of one of the two folded subgraphs, and the nested dissection process car recursively proceed independently on each subgroup of $\frac{p}{2}$ (then $\frac{p}{4}$, $\frac{p}{8}$, etc.) processors, until each subgroup is reduced to a single processor. From then on, the nested dissection process will go on sequentially on every processor, using the nested dissection routines of the Scotch library, eventually ending in a coupling with minimum degree methods [6] (which are thus only used in a sequential context).

3.2. Graph coarsening

The second level of concurrency concerns the computation of separators. The approach we have chosen is the now classical multi-level one [7–9]. It consists in repeatedly computing a set of increasingly coarser albeit topologically similar versions of the graph to separate, by finding matchings which collapse vertices and edges, until the coarsest graph obtained is no larger than a few hundreds of vertices, then computing a separator on this

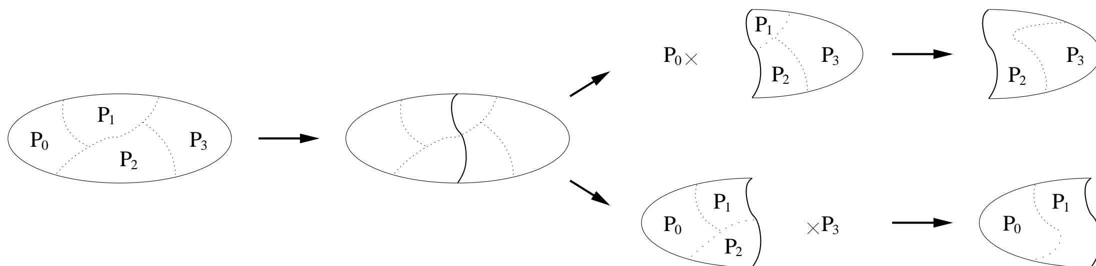

Figure 2. Diagram of a nested dissection step for a (sub-)graph distributed across four processors. Once the separator is known, the two induced subgraphs are built and folded (this can be done in parallel for both subgraphs), yielding two subgraphs, each of them distributed across two processors.

coarsest graph, and projecting back this separator, from coarser to finer graphs, up to the original graph. Most often, a local optimization algorithm, such as Kernighan-Lin [10] or Fiduccia-Mattheyses [11] (FM), is used in the uncoarsening phase to refine the partition that is projected back at every level, such that the granularity of the solution is the one of the original graph and not the one of the coarsest graph.

The main features of our implementation are outlined in Figure 3. The matching of vertices is performed in parallel by means of a synchronous probabilistic algorithm. Every processor works on a queue storing the yet unmatched vertices which it owns, and repeats the following steps. The queue head vertex is dequeued, and a candidate for mating is chosen among its unmatched neighbors, if any; else, the vertex is left unmatched at this level and discarded. If the candidate vertex belongs to the same processor, the mating is immediatly recorded, else a mating request is stored in a query buffer to be sent to the proper neighbor processor, and both vertices (the local vertex and its ghost neighbor) are flagged as temporarily unavailable. Once all vertices in queue have been considered, query buffers are exchanged between neighboring processors, and received query buffers are processed in order to satisfy feasible pending matings. Then, unsatisfied mating requests are notified to their originating processors, which unlock and reenqueue the unmatched vertices. This whole process is repeated until the list is almost empty; we do not wait until it is completely empty because it might require too many collective steps for just a few remaining vertices. It usually converges in 5 iterations.

Once the matching phase is complete can the coarsened subgraph building phase take place. This latter can be parametrized so as to allow one to choose between two options. Either all coarsened vertices are kept on their local processors (that is, processors that hold at least one of the ends of the coarsened edges), as shown in the first steps of Figure 3, which decreases the number of vertices owned by every processor and speeds-up future computations, or else coarsened graphs are folded and duplicated, as shown in the next steps of Figure 3, which increases the number of working copies of the graph and can thus reduce communication and increase the final quality of the separators.

As a matter of fact, separator computation algorithms, which are local heuristics, heav-

ily 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 ones, obtained from two fully independent multi-level runs, usually improved 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 processors that hold a working copy of the graph being able to perform an almost-complete independent multi-level computation, save for the very first level which is shared by all subgroups, for the second one 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 processor 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 processor is about twice the size of the original graph. When folding occurs, every processor receives two coarsened parts, one of which belonging to another processor, such that the size of the folded part is about the one of the finer graph. The footprint of the fold-dup scheme is therefore logarithmic in the number of processors, and may consume all available memory as this number increases. Consequently, as in [12], 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 a trade off between the level of completeness of the independent multi-level 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.

Once all working copies of the coarsened graphs are folded on individual processors, the algorithm enters a multi-sequential phase, illustrated at the bottom of Figure 3: the routines of the sequential Scotch library are used on every processor to complete the coarsening process, compute an initial partition, and project it back up to the largest centralized coarsened graph stored on the processor. Then, the partitions are projected back in parallel to the finer distributed graphs, selecting the best partition between the two available when projecting to a level where fold-dup had been performed. This distributed projection process is repeated until we obtain a partition of the original graph.

3.3. Band refinement

The third level of concurrency concerns the refinement heuristics which are used to improve the projected separators. At the coarsest levels of the multi-level algorithm, when computations are restricted to individual processors, the sequential FM algorithm of Scotch is used, but this class of algorithms does not parallelize well. Indeed, a parallel FM-like algorithm has been proposed in ParMeTiS [1] but, in order to relax the strong sequential constraint that would require some communication every time a vertex to be migrated has neighbors on other processors, only moves that strictly improve the partition are allowed, which hinders the ability of the FM algorithm to escape local minima of its cost function, and leads to severe loss of partition quality when the number of processors (and thus of potential remote neighbors) increase.

This problem can be solved in two ways: either by developing scalable and efficient local optimization algorithms, or by being able to use the existing sequential FM algorithm on

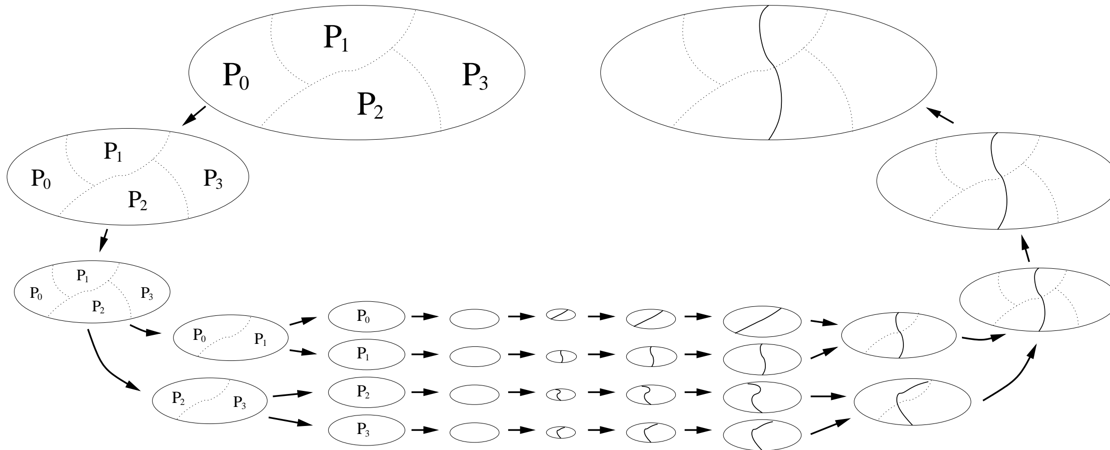

Figure 3. Diagram of the parallel computation of the separator of a graph distributed across four processors, by parallel coarsening with folding-with-duplication, multi-sequential computation of initial partitions that are locally projected back and refined on every processor, and then parallel uncoarsening of the best partition encountered.

very large graphs. We have proposed and successfully tested in [13] a solution which enables both approaches, and is based on the following reasoning. Since every refinement is performed by means of a local algorithm, which perturbs only in a limited way the position of the projected separator, local refinement algorithms need only to be passed a subgraph that contains the vertices that are very close to the projected separator. We have experimented that, when performing FM refinement on band graphs that contain only vertices that are at distance at most 3 from the projected separators, the quality of the finest separator does not only remain constant, but even significantly improves in most cases. Our interpretation is that this pre-constrained banding prevents the local optimization algorithms from exploring and being trapped in local optima that would be too far from the global optimum sketched at the coarsest level of the multi-level process.

The advantage of pre-constrained band FM is that band graphs are of a much smaller size than their parent graphs, since for most graphs the size of the separators is of several orders of magnitude smaller that the size of the separated graphs: it is for instance in $O(n^{\frac{1}{2}})$ for 2D meshes, and in $O(n^{\frac{2}{3}})$ for 3D meshes [14]. Consequently, FM or other algorithms can be run on graphs that are much smaller, without decreasing separation quality.

The computation and use of distributed band graphs is outlined in Figure 4. Given a distributed graph and an initial separator, which can be spread across several processors, vertices that are closer to separator vertices than some small user-defined distance are selected by spreading distance information from all of the separator vertices, using our halo exchange routine. Then, the distributed band graph is created, by adding on every processor 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.

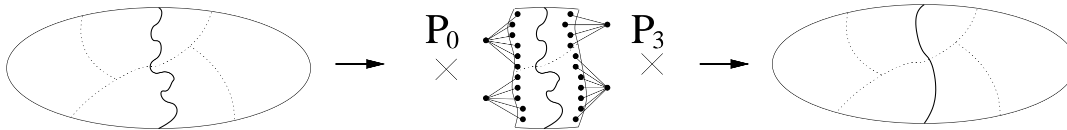

Figure 4. Creation of a distributed band graph. Only vertices closest to the separator are kept. Other vertices are replaced by anchor vertices of equivalent total weight, linked to band vertices of the last layer. There are two anchor vertices per processor, to reduce communication. Once the separator has been refined on the band graph using some local optimization algorithm, the new separator is projected back to the original distributed graph.

Once the separator of the band graph has been refined using some local optimization algorithm, the new separator is projected back to the original distributed graph.

Basing on our band graphs, we have implemented a multi-sequential refinement algorithm, outlined in Figure 5. At every distributed uncoarsening step, a distributed band graph is created. Centralized copies of this band graph are then gathered on every participating processor, which serve to run fully independent instances of our sequential FM algorithm. The perturbation of the initial state of the sequential FM algorithm on every processor allows us to explore slightly different solution spaces, and thus to improve refinement quality. Finally, the best refined band separator is projected back to the distributed graph, and the uncoarsening process goes on.

Centralizing band graphs is an acceptable solution because of the much reduced size of the band graphs that are centralized on the processors. Using this technique, we expect to achieve our goal, that is, to be able partition graphs up to a billion vertices, distributed on a thousand processors,without significant loss in quality, because centralized band graphs will be of a size of a few million vertices for 3D meshes. In case the band graph cannot be centralized, we can resort to a fully scalable algorithm, as partial copies can also be used collectively to run a scalable parallel multi-deme genetic optimization algorithm, such as the one experimented with in [13].

4. Experimental results

PT-SCOTCH is written in ANSI C, with calls to the POSIX thread and MPI APIs. The most significant test graphs that we have used in our tests are presented in Table 1. All of our experiments were performed on the M3PEC system of Université Bordeaux 1, an IBM cluster made of SMP nodes comprising 8 dual-core Power5 processors running at 1.5 GHz.

All of the ordering strategies that we have used were based on the multi-level scheme. During the uncoarsening step, separator refinement was performed by using our sequential FM algorithm on band graphs of width 3 around the projected separators, both in the parallel (with multi-centralized copies) and in the sequential phases of the uncoarsening process.

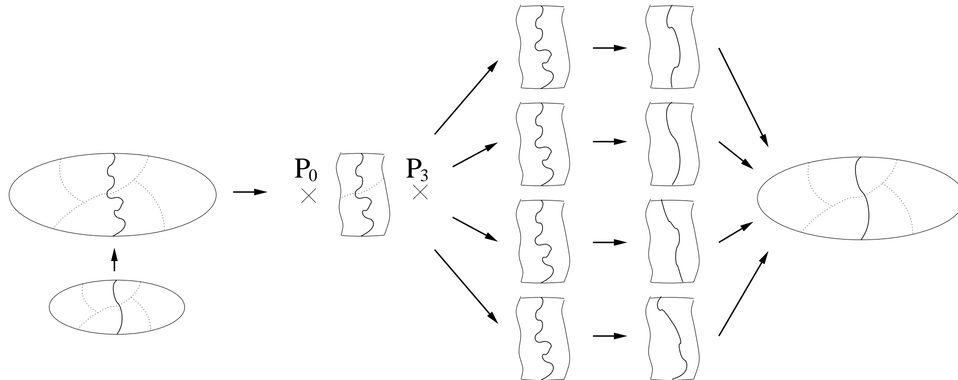

Figure 5. Diagram of the multi-sequential refinement of a separator projected back from a coarser graph distributed across four processors to its finer distributed graph. Once the distributed band graph is built from the finer graph, a centralized version of it is gathered on every participating processor. A sequential FM optimization can then be run independently on every copy, and the best improved separator is then distributed back to the finer graph.

The quality criterion that we have chosen is the operation count (OPC) required to factor the reordered matrix using the Cholesky method; it is an indirect measurement of the overall quality of all bipartitions, in the practical context of nested dissection ordering.

Table 2 presents the OPC computed on the orderings yielded by PT-Scotch and ParMeTiS. These results have been obtained by running PT-Scotch with the following strategy: in the multi-level 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 processors and the sequential multi-level process relays it.

The improvement in quality brought by PT-Scotch is clearly evidenced, and increases along with the number of processes, as our local optimization scheme is not sensitive to this number. While PT-Scotch is, at the time being, about three times slower on average than ParMeTiS, it can yield operation counts that are as much as two times smaller than the ones of ParMeTiS, which is of interest as factorization times are more than one order of magnitude higher than ordering times. Most of the time we consume is spent in the coarsening phase, the efficiency and scalability of which has yet to be improved.

In order to evidence the memory overhead of the fold-dup process, we have run two tests with different folding strategies. As said above, both strategies end up by performing fold-dup when the average number of vertices per processor falls under 100, but they differ in the following way: in the first one, referred to as NoFD, no folding is performed in the first coarsening stages at all; in the second one, called AllFD, fold-dup is performed at every step from the very first one, such that after $\log_2(p)$ steps each of the p processors holds a centralized copy of the graph, onto which the sequential algorithm is run.

Graph	Size ($\times 10^3$)		Average degree	O_{SS}
	$\|V\|$	$\|E\|$		
altr4	26	163	12.50	3.65e+8
audikw1	944	38354	81.28	5.48e+8
bmw32	227	5531	48.65	2.75e+8
conesphere1m	1055	8023	15.21	1.83e+12
coupole8000	1768	41657	47.12	7.48e+10
oilpan	74	1762	47.77	2.74e+9
thread	30	2220	149.32	4.14e+10

Table 1
Description of some of the test graphs that we use. $|V|$ and $|E|$ are the vertex and edge cardinalities, in thousands, and O_{SS} is the operation count of the Cholesky factorization performed on orderings computed using the sequential Scotch software.

The factorization results presented in table 3 show that both strategies are very close in term of OPC. Therefore, fully independent multi-level runs are not mandatory to achieve good ordering quality, which is hopeful as the AllFD strategy consumes a lot more memory than NoFD. It can however be noted that, because of the higher number of runs that it performs, the output of AllFD is more predictible, *i.e.* there are less variations in quality due to execution factors such as randomness.

The high memory consuption of the AllFD strategy is clearly evidenced in Table 4. The average amount of memory per processor needed by NoFD ranges between a half and a fourth of the one required by AllFD. These values can be easily explained by the cost of duplicating data at each fold-dup step. Indeed, to perform fold-dup on the first k levels requires $\Theta(k)$ times the amount of memory to store the initial graph, whereas performing simple coarsening requires only $\Theta\left(\sum_{i=1}^{k} \frac{1}{2^i}\right) = \Theta(2)$ times this initial amount of memory. Nevertheless, for smaller graphs like **altr4** or **thread**, both values are very similar for 64 processors, since then the NoFD strategy performs fold-dups from the third coarsening level.

Table 5, which contains the time measurements of both strategies, shows that the AllFD strategy is often the fastest when the number of processors increases. Only does graph **coupole8000** always have better times with the NoFD strategy, because it has enough vertices to cover communication latency. These results clearly show that the coarsening process is critical in terms of communication efficiency. The AllFD strategy is faster because early folding processes gather the vertices of the coarsened graphs onto a smaller number of processors, which increases the ratio of local end vertices in the mating process and reduces the amount of communication to be performed, both in number of messages and in message size. This is why running times explode for smaller graphs (all of them except **coupole8000**) when running NoFD with 64 processors. Until the coarsening process is not improved, a good strategy could therefore be to resort to fold-dup when the number of vertices is still high, about 10000 per processor.

5. Conclusion

We have presented in this paper the parallel algorithms that we have implemented in PT-Scotch to compute in parallel efficient orderings of large graphs. The first results are encouraging, as they meet the expected performance requirements in term of quality, but have to be improved in term of scalability.

Memory consumption is clearly a problem to be solved. This can be done by not keeping the `edgeloctab` of intermediate graphs, as it is not needed by most algorithms. Therefore, the `edgegsttab` could replace the `edgeloctab` in place, without any memory overhead. This is currently under development.

Although it corresponds to a current need within the ScAlApplix project, to obtain as quickly as possible high quality orderings of graphs with a size of a few tens of millions of vertices, sparse matrix ordering is not the application field in which we expect to find the largest problem graphs, as existing parallel direct sparse linear system solvers cannot currently handle full 3D meshes of a size larger than about fifty million unknowns.

Therefore, basing on the software building blocks that we have already written, we plan to extend the capabilities of PT-Scotch to compute k-ary edge partitions of large meshes for subdomain-based iterative methods, as well as static mappings of process graphs, as the Scotch library does sequentially.

REFERENCES

1. MeTiS: Family of multilevel partitioning algorithms, `http://glaros.dtc.umn.edu/gkhome/views/metis`.
2. Jostle: Graph partitioning software, `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.
3. W. F. Tinney, J. W. Walker, Direct solutions of sparse network equations by optimally ordered triangular factorization, J. Proc. IEEE 55 (1967) 1801–1809.
4. T.-Y. Chen, J. R. Gilbert, 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.
5. A. George, J. W.-H. Liu, Computer solution of large sparse positive definite systems, Prentice Hall, 1981.
6. F. Pellegrini, J. Roman, P. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, Concurrency: Practice and Experience 12 (2000) 69–84.
7. S. T. Barnard, H. D. Simon, A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency: Practice and Experience 6 (2) (1994) 101–117.
8. B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of Supercomputing, 1995.
9. G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.
10. B. W. Kernighan, S. Lin, An efficient heuristic procedure for partitionning graphs, BELL System Technical Journal (1970) 291–307.

11. C. M. Fiduccia, R. M. Mattheyses, A linear-time heuristic for improving network partitions, in: Proceedings of the 19th Design Automation Conference, IEEE, 1982, pp. 175–181.
12. G. Karypis, V. Kumar, ParMetis Parallel Graph Partitioning and Sparse Matrix Ordering Library, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A. (August 2003).
13. C. Chevalier, F. Pellegrini, Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework, in: Proc. Europar, Dresden, LNCS 4128, 2006, pp. 243–252, `http://www.labri.fr/~pelegrin/papers/scotch_efficientga.pdf`.
14. R. J. Lipton, D. J. Rose, R. E. Tarjan, Generalized nested dissection, SIAM Journal of Numerical Analysis 16 (2) (1979) 346–358.

Test case	**Number of processors or threads**					
	2	4	8	16	32	64
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}	†	†	7.78e+12	8.88e+12	8.91e+12	1.07e+13
t_{PTS}	105.48	102.42	96.48	77.65	70.88	108.89
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
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
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
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 2
Comparison between ParMeTiS (PM) and PT-Scotch (PTS) for several graphs. O_{PTS} and O_{PM} are the OPC for PTS and PM, respectively. Dashes indicate abortion due to memory shortage. Daggers indicate erroneous output permutations. ParMeTiS yielded erroneous permutations for all numbers of processors with graph **oilpan**.

Strategy case	**Number of processors**					
	2	4	8	16	32	64
altr4 – $OPC_{seq} = 3.65e+8$						
NoFD	3.84e+8	**3.75e+8**	3.93e+8	**3.69e+8**	4.09e+8	**4.15e+8**
AllFD	**3.75e+8**	3.92e+8	**3.80e+8**	3.77e+8	**3.88e+8**	4.16e+8
audikw1 – $OPC_{seq} = 5.48e+12$						
NoFD	5.73e+12	**5.65e+12**	**5.54e+12**	**5.45e+12**	**5.45e+12**	**5.45e+12**
AllFD	**5.69e+12**	5.67e+12	5.96e+12	5.86e+12	5.88e+12	–
bmw32 – $OPC_{seq} = 2.75e+10$						
NoFD	3.50e+10	**3.49e+10**	**3.14e+10**	**3.05e+10**	**3.02e+10**	3.00e+10
AllFD	**3.14e+10**	3.62e+10	3.67e+10	3.29e+10	3.04e+10	**2.80e+10**
conesphere1m – $OPC_{seq} = 1.83e+12$						
NoFD	**1.88e+12**	**1.89e+12**	**1.85e+12**	**1.84e+12**	1.86e+12	**1.77e+12**
AllFD	2.04e+12	2.85e+12	2.39e+12	1.87e+12	**1.80e+12**	1.79e+12
coupole8000 – $OPC_{seq} = 7.48e+10$						
NoFD	8.68e+10	8.54e+10	8.38e+10	**8.03e+10**	**8.26e+10**	**8.21e+10**
AllFD	**8.66e+10**	**8.53e+10**	**8.09e+10**	8.20e+10	8.25e+10	–
oilpan – $OPC_{seq} = 2.74e+9$						
NoFD	4.08e+9	3.79e+9	3.31e+9	3.99e+9	3.29+9	3.59e+9
AllFD	**3.45e+9**	**3.38e+9**	**3.19e+9**	**3.77e+9**	**3.28e+9**	**3.30e+9**
thread – $OPC_{seq} = 4.14e+10$						
NoFD	**3.52e+10**	4.31e+10	4.13e+10	**4.06e+10**	**4.06e+10**	4.50e+10
AllFD	4.20e+10	**4.01e+10**	**3.97e+10**	4.17e+10	4.25e+10	**4.03e+10**

Table 3
OPC of orderings computed by PT-Scotch with two different folding strategies. Dashes indicate abnormal termination due to memory shortage.

Strategy case	**Number of processors**					
	2	4	8	16	32	64
altr4						
NoFD	**4.51e+6**	**3.06e+6**	**1.91e+6**	**1.46e+6**	**1.31e+6**	**1.24e+6**
AllFD	7.55e+6	5.42e+6	3.85e+6	2.64e+6	1.75e+6	1.29e+6
audikw1						
NoFD	**9.18e+8**	**6.54e+8**	**4.05e+8**	**2.37e+8**	**1.37e+8**	**7.99e+7**
AllFD	1.42e+9	1.16e+9	8.77e+8	6.26e+8	4.37e+8	–
bmw32						
NoFD	**1.21e+8**	**8.35e+7**	**5.35e+7**	**3.23e+7**	**1.91e+7**	**1.33e+7**
AllFD	1.51e+8	1.08e+8	7.69e+7	5.07e+7	3.32e+7	2.29e+7
conesphere1m						
NoFD	**2.38e+8**	**1.54e+8**	**9.75e+7**	**5.83e+7**	**3.44e+7**	**2e+7**
AllFD	3.91e+8	2.91e+8	2.17e+8	1.54e+8	1.09e+8	7.51e+7
coupole8000						
NoFD	**1.04e+9**	**6.97e+8**	**4.28e+8**	**2.58e+8**	**1.51e+8**	**8.67e+7**
AllFD	1.67e+9	1.07e+9	7.06e+8	4.43e+8	2.74e+8	–
oilpan						
NoFD	**3.86e+7**	**2.65e+7**	**1.7e+7**	**1.02e+7**	**6.1e+6**	**3.57e+6**
AllFD	4.93e+7	3.43e+7	2.32e+7	1.48e+7	1.04e+7	6.94e+6
thread						
NoFD	**4.91e+7**	**3.74e+7**	**2.37e+7**	**1.66e+7**	**1.31e+7**	**1.18e+7**
AllFD	7.31e+7	5.98e+7	4.44e+7	3.2e+7	2.17e+7	1.35e+7

Table 4
Average memory usage per processor, in bytes, when running of PT-Scotch two different folding strategies. Dashes indicate abnormal termination due to memory shortage.

Strategy case	Number of processors					
	2	4	8	16	32	64
altr4						
NoFD	**0.42**	**0.30**	**0.24**	**0.30**	0.52	1.55
AllFD	**0.42**	0.31	0.29	0.33	**0.39**	**0.76**
audikw1						
NoFD	105.48	102.42	96.48	77.65	70.88	**108.89**
AllFD	**70.86**	**70.53**	**75.14**	**75.90**	**69.82**	–
bmw32						
NoFD	8.89	7.41	5.68	5.45	8.36	17.64
AllFD	**7.73**	**6.49**	**5.59**	**5.38**	**5.82**	**8.05**
conesphere1m						
NoFD	31.34	**20.41**	**18.76**	**18.37**	25.80	92.47
AllFD	**31.32**	22.27	20.19	18.69	**19.06**	**23.64**
coupole8000						
NoFD	114.41	116.83	85.80	**60.23**	**41.60**	**28.10**
AllFD	**86.11**	**93.78**	**85.23**	70.33	54.81	–
oilpan						
NoFD	2.53	1.81	**1.33**	**1.10**	1.51	3.75
AllFD	**2.05**	**1.65**	1.39	1.31	**1.49**	**2.23**
thread						
NoFD	3.66	3.61	3.30	**3.65**	**5.68**	11.16
AllFD	**2.45**	**2.89**	**3.22**	3.99	5.76	**9.03**

Table 5
Ordering times of PT-Scotch (in seconds) when performing two different folding strategies. Dashes indicate abnormal termination due to memory shortage.