

A graph based framework for the definition of tools dealing with sparse and irregular distributed data structures*

Serge Chaumette, Jean-Michel Lépine and Franck Rubi
LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I,
351 Cours de la Libération,
33405 Talence, France

{Serge.Chaumette, Jean-Michel.Lepine, Franck.Rubi}@labri.u-bordeaux.fr

Abstract

Industrial applications use specific problem-oriented implementations of large sparse and irregular data structures. Hence there is a need for tools that make it possible for developers to visualize their applications in terms of these data, their structure, and the operations that are applied to them, whatever their effective implementation and their distribution are. In this paper, we present both a framework which we have setup to support the development of such tools, and prototypes which we have developed. The resulting environment is composed of two layers: the first layer is a model that we have defined and implemented as high level libraries that make it possible to efficiently abstract from the implementation; the second layer offers prototype tools built on top of these libraries. These tools are integrated within a graphical environment called Visit[7] which is part of the HPFIT research effort¹ [7, 8, 9]. HPFIT is a joint project involving three research laboratories: LIP in Lyon, France, LaBRI in Bordeaux, France, and GMD/SCAI in Bonn, Germany. Its aim is to provide an integrated HPF development environment that supports sparse and irregular data structures.

1. Introduction

Industrial applications use standard data structures such as matrices, but most of the time provide a specific problem-oriented implementation, e.g. Compressed Sparse Column (CSC) – see for instance SPARSKIT[23]. Specific implementations are often used when dealing with large sparse and irregular data structures, such as matrices coming from

the domain of finite elements[25]. The gap between the implementation and the abstract data structure it implements is even bigger when considering data-parallel applications where data structures are distributed over a network of processors. Hence there is a need for tools that make it possible for developers to visualize both their data, their structure, and the operations that are applied to it, whatever their effective implementation and their distribution are. Such tools must provide high level views, i.e. abstract from the physical implementation to reach the developer’s view. They must carry the semantics of the applications and provide synthesis or filtering mechanisms that make it possible to focus on a specific aspect of the problem.

2. Related work

The two main research directions within the data-parallel framework are **expression** modes (or languages) and **tools**.

Expression is widely studied. The aim of many initiatives is either to introduce sparse and irregular data structures in data-parallel languages (see [4] and [5]) or to develop dedicated libraries, to provide programmers with ready-to-use implementations of possibly sparse and distributed data structures (see for instance SPARSEKIT[23], P-SPARSLIB[24], PETSc[22] or TNT[21]).

Tools exist, but they mainly deal with **regular** (HPF-like) data structures (see for instance EPPP[16], P2D2[13], Pharos[26] and Paradyn[19]). We will not describe all of them in this paper because they are less related to our goal which is to deal with irregular data structures (see [20] for a complete survey).

Concerning **irregularity** there is still a lot to be done in terms of tools that would help users to tackle the paradigm of data-parallel programming. Nevertheless, the research

*This work is supported by the French GDR-PRC PRS.

¹This work is partly supported by the CNRS-INRIA project ReMaP and by the French GDR-PRC PRS.

which has been done during past years in the area of message passing has proven quite successful in providing support to end-users (see for instance TOPSYS[2, 3, 6]). Both hardware and software vendors supply environments of their own. Furthermore, public domain tools (such as ParaGraph[15]) are now being delivered within vendors environments. These tools that were used when dealing with message-passing applications can still be used within the framework of data parallelism provided the execution support is distributed (in which case the compiler translates data parallelism to message passing parallelism). Nevertheless, there is a lack of relationship between the information they provide (which is in terms of messages) and the semantics of the application (which is in terms of data). This makes them hardly usable to explore the algorithmic behavior of the application, although they can still be useful for performance measurement. The reason for this lack of adequation is that the behavior of an algorithm is most of the time better understood when considering the *abstract structure* it works with, rather than the *physical implementation* of this data structure. For instance if the algorithm handles a tree that is implemented using a vector, it is most probably the case that this algorithm can be better understood in terms of the tree than in terms of the vector. One of the reasons why high level tools are missing is that they require information that cannot always be accessed easily, such as distribution of data. A convenient way to deal with this problem is to rely on the user to supply these information. This is the approach which is for instance implemented in IVD[14]. This is always quite heavy for the programmer. Another manner is to have libraries that “instrument” the basics of the language and which are linked to the application at the same time as the language libraries themselves. This is the approach which is achieved in one of PTOOLS projects called *Distributed Array Query and Visualization* (DAQV [18]). This is not straightforwardly portable.

3. Aim of this work

Compared to those described above, our **approach** is quite different. **Our aim is not to provide support for using sparse and irregular data-structures inside applications: we want to provide tools dealing with such data structures at a high level of abstraction. Furthermore, we want to make it easy to develop such tools.** Hence, we first designed libraries to abstract from implementation and distribution of data structures. These libraries are portable and rely very little on the programmer. We then developed tools based on these libraries. In this paper we present both the concepts of these libraries and the tools based on them. These tools are integrated within a graphical environment called Visit[7] which is part of the HPFIT research effort[7, 8, 9].

The rest of this paper is organized as follows. We first describe the overall architecture of the model we propose in section 4. Section 5 introduces the way we modelize data structures. We then describe the two current levels of our model in sections 6, 7 and 8. Section 9 presents the general principles of the tools based on our model. It is illustrated with two software components which we have developed: *Data Distribution Display* and *Trace Data Display*. We eventually sketch future work directions.

4. General architecture of the model

Our model is composed of three levels. Each of these levels, i.e. implementation, abstraction and view is a graph (figure 1). A mapping is defined to describe the relationship between these levels. It is a bridge that carries the semantics between the different levels.

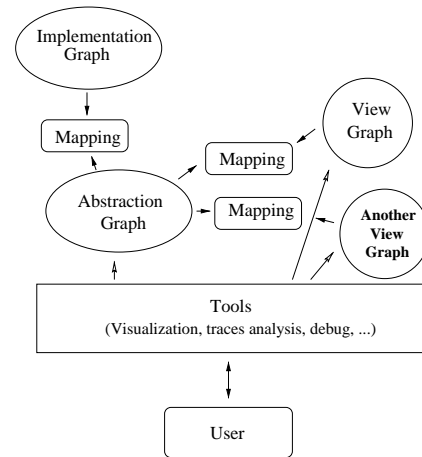


Figure 1. Architecture of the model

1. IGraph: the Implementation Graph (figure 2) describes the implementation of the data structure, in terms of data items and access functions, i.e. the way they are accessed within the application – e.g. three vectors for a Compressed Sparse Column Storage.
2. AGraph: the Abstraction Graph (figure 3) describes the abstract data structure the application developer has in mind – e.g. a matrix.
3. VGraph: the View Graph (figures 4 and 5) describes how a tool will eventually “see” the data structure – e.g. a column of a matrix. This level provides for synthesis and filtering of information.

It is not yet implemented (see section 10), but one can work with the AGraph, which is equivalent to having an identity between the AGraph and the VGraph.

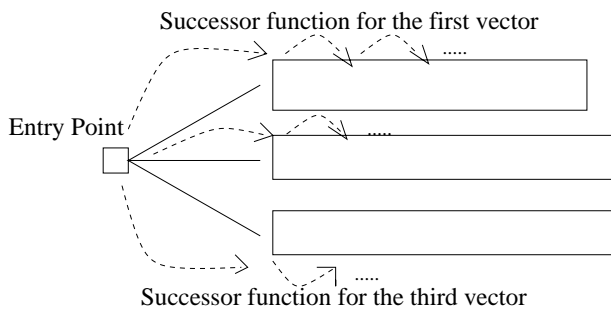


Figure 2. Implementation

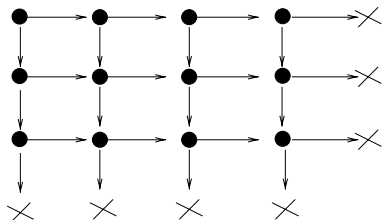


Figure 3. Abstraction as a matrix

Although this will not be detailed here, the model provides the same efficiency when working at the mapping level or at the implementation graph level (see figure 6).

5. Modelization of a data structure

We define a data structure in terms of entry points and access functions. This approach reflects the way a data structure is accessed within an application, i.e. the way it is implemented. It leads to a modelization in terms of a graph. For instance a vector `int v[10]` can be represented by a graph, the root of which is `v[0]` and the nodes are `v[i]`.

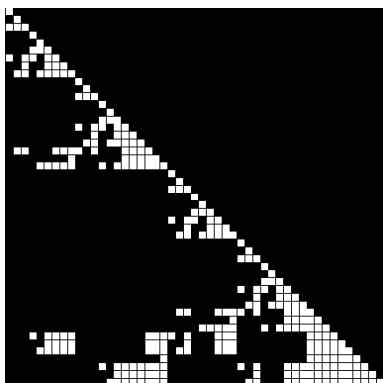


Figure 4. View as a matrix

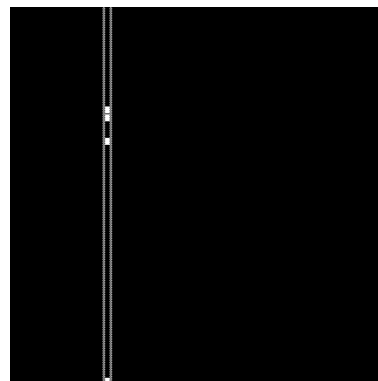


Figure 5. View as a column

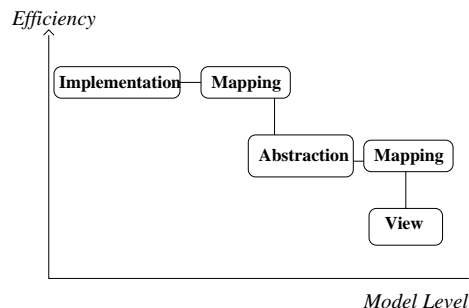


Figure 6. Efficiency of the model levels

The access functions that express how one moves around the data structure implement the neighbouring relationship between data items, e.g. the neighbour of `v[i]` is `v[i+1]`.

Definition 1 Data structure

A data structure DS is a set of data items that are structured by means of access functions.

$$DS = (D, d, F, e, E)$$

where :

- D is the set of data items;
- d is a positive integer representing the number of access functions;
- F is a d -uple of access functions;
- e is a positive integer representing the number of entry points;
- E is the set of entry points in the data structure.

In other words, a data-structure is a graph, the nodes of which contain the data items, the edges of which represent access functions.

Any data structure, even irregular, can be described using this framework, since the nodes of the data structure graph can themselves be data structures, i.e. graphs.

Access functions f_i express how one accesses the basic data items. Note that for the same implementation there might be different sets of access functions, depending on how the programmer accesses the data structure.

Definition 2 (Neighbouring) Access function
 An **access function** is a function over the nodes of the data structures that being given a node produces another node.

$$f : N \rightarrow N$$

$$n \mapsto f(n)$$

Each access function makes it possible to move within a **dimension** of the data structure.

We call a *direct-access data structure*, a data structure in which each item can be directly accessed (e.g. a set).

Definition 3 Direct-access data structure
 A data structure $DS = (D, d, F, e, E)$ is said **direct access** if and only if $E = D$, $d = 0$, $F = \emptyset$ and $e = |D|$.

In such a case all of the data items of the data structure are roots of the graph used to describe it.

In [7], we introduce further definitions based on previous work by M. Alabau[1], that provide matrix-like notations for multi-dimensional objects represented by graphs.

6. Describing the implementation data structure

In this section we present an example, the aim of which is to illustrate how the definitions of section 5 can be used to describe an effective implementation.

The implementation is composed of three vectors of different sizes (figure 7). An entry point is added to obtain a connected graph. The nodes are the values of the three vectors plus the entry point. We give three successor functions; each function allows to access the nodes of one of the three vectors.

This graph may match a Compressed Sparse Column implementation of a sparse matrix. It may also be interpreted as a Compressed Sparse Row implementation, or as any other data structure. The interpretation of this implementation as a graph does not describe the semantics of what is effectively implemented. The benefit is that the description of this graph by the programmer is straightforward.

Figure 8 shows the code the programmer must write in order to describe this implementation.

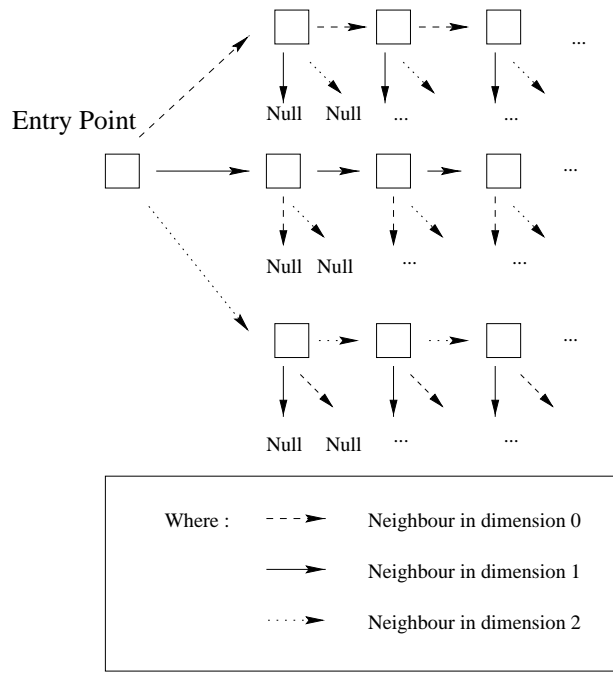


Figure 7. A graph modelization of the implementation

7. Describing the abstraction data structure

We call *abstraction* the abstract data structure that the user wants to manipulate. Since an abstraction is itself a data structure, it can be represented by an entry point and access functions, i.e. a graph. The aim of the abstraction is to offer an interpretation of the implementation. For instance the three vectors of section 6 can be interpreted as matrix (see figure 9).

Figure 10 shows the code the programmer must write in order to describe this abstraction.

Abstracting from an implementation then consists in mapping the implementation graph to the abstraction graph.

8. Mapping

The mapping establishes a correspondence between the implementation graph and the abstraction graph. It is used to move from the abstraction to the implementation and vice-versa.

It is a set of pairs containing both a node of the implementation graph and a node of the abstraction graph. Such a node can only exist provided there is a node of the implementation corresponding to the given node of the abstraction (the abstraction which is dense usually has more nodes

```

1  /*****
2  IMPLEMENTATION NODE INDEX DEFINITION
3  *****/
4  extern IDS ids; /* The Irregular Data Structure */
5  /* composed here of 3 Vectors of different sizes */
6  /*-----*/
7  struct IDSIndex{
8  int d0; /* an index for the Irregular Data Structure, a value */
9  int d1; /* di (i=0, 1 or 2) gives a position in the */
10 int d2; /* corresponding vector */
11 int d2;
12 };
13 /*-----*/
14 /* function used to create a new IDSIndex */
15 IDSIndex idsindex_new(int v0, int v1, int v2){
16 ... allocate the idsindex and copy v0, v1, v2 to d0, d1, d2 ...
17 return IDSIndex;
18 }
19 /*****
20 IMPLEMENTATION GRAPH DEFINITION (IGraph)
21 *****/
22 /* function used to give the index root */
23 static IDSIndex getRoot(IGraph iGraph){
24 return idsindex_new(-1,-1,-1);
25 }
26 /*-----*/
27 /* this function returns the next node if exists in dimension dim */
28 static IDSIndex next( int dim, IDSIndex IDSIndex){
29 /* => Except for root nodeindex , one di is different of -1 */
30 /* its unique neighbour is in dimension i */
31 /* at the end of a vector, we return NULL value */
32 switch (dim) {
33 case 0:
34 if ((IDSIndex->d1 != -1) || (IDSIndex->d2 != -1)) return NULL;
35 if (IDSIndex->d0 < IDS->V0_size-1)
36 return idsindex_new(IDSIndex->d0+1,IDSIndex->d1, IDSIndex->d2);
37 break;
38 case 1: ... the same as case 0 but for dimension 1 or 2 ...
39 case 2:
40 }
41 return NULL;
42 }
43 /*-----*/
44 /* this function returns for an IDSIndex the corresponding value */
45 static void * getNodeValue( IDSIndex IDSIndex){
46 /* only one di (i = 0,1 or 2) has a value different from -1 */
47 if (IDSIndex->d0 != -1) return (void *) &(IDS->V0[IDSIndex->d0]);
48 if (IDSIndex->d1 != -1) return (void *) &(IDS->V1[IDSIndex->d1]);
49 if (IDSIndex->d2 != -1) return (void *) &(IDS->V2[IDSIndex->d2]);
50 return NULL;
51 }
52 /*-----*/

```

Figure 8. IGraph description

than the implementation which is sparse). Assume a function that, given a node of the abstraction, returns either a node of the implementation or NULL (probably a hole in the sparse data structure). Now on we will call this function *abs_to_imp*.

We can then define the nodes of the mapping graph as follows:

Definition 4 $Nodes = \{(abs_to_imp(anode), anode) \mid anode \in \text{a graph nodes}\}$

Remark We can also define a function *imp_to_abs* (implementation node to abstraction node). It allows for another symmetric definition of the mapping graph. The user can define these two functions (to gain efficiency). One of these functions may be difficult to write: it can be left to

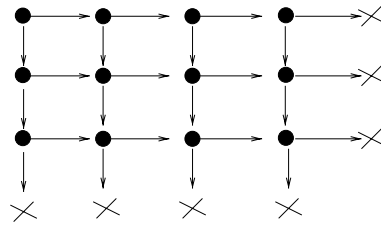


Figure 9. Abstraction as a dense matrix

the tools to construct it using the other function, storing this information into a hashtable.

Figure 11 shows the code the programmer must write in order to describe this mapping.

To allow an efficient scanning of set of pairs of the mapping, it is possible to define a root and a next function. The next function orders of the mapping pairs, according to an efficient scanning of the implementation.

9. Usage

The framework presented above is being used to develop high level tools. In this section we present two prototypes.

9.1. General principle

Using our libraries, high level tools work only with graphs, which they can, for instance, scan, using either of the loops shown in figures 12 and 14.

Hence, implementing tools becomes straightforward, since they mainly deal with data structures the way they would do using standard multidimensionnal arrays as shown in figure 13.

Furthermore, the library makes it possible to attach additional information to any node of any of the graphs. This is used for instance by the traces visualization tool described below, to store internal calculation results.

9.2. Effective usage

The goal of our tools is to offer a set of software components to visualize and analyze the behavior of data-parallel programs in terms of the data they work with. The framework presented above is being used to develop these tools. They are implemented either in terms of abstraction graphs, mapping or view graphs.

As of writing, some of these tools are available as prototypes. This section describes two of them: **DDD** (Data Distribution Display) and **TDD** (Trace Data Display). Parts of these tools are components of the HPFIT project. A complete description can be found in [12]. We only present

```

1  /*****
2  ABSTRACTION NODE INDEX (AGraph)
3  *****/
4  /* an ANodeIndex of a node corresponds to a path between the root node */
5  /* and the node in terms of ordered moves */
6  /* the ANodeIndex of the root node is an empty path */
7  #include <aNodeIndex.h>
8
9  /*****
10 ABSTRACTION GRAPH DEFINITION (AGraph)
11 *****/
12 /* function used to give the index root */
13 static ANodeIndex getRoot(AGraph aGraph){
14     return anodeindex_new(); /* return an empty ANodeIndex */
15 }
16 /*-----*/
17 /* this function returns the next node if exists in dimension dim */
18 /* <=> matrix[i,j] to matrix[i+1,j] or matrix[i,j+1] */
19 static ANodeIndex next(AGraph aGraph, int dimension, ANode aNodeIndex){
20
21     extern int nbRows /* number of lines */
22     extern int nbCols /* number of columns */
23
24     /* i : the number of moves in the dimension 0 (rows) */
25     int i = anodeindex_depth(aNodeIndex, 0);
26     /* j : the number of moves in the dimension 1 (columns)*/
27     int j = anodeindex_depth(aNodeIndex, 1);
28
29     switch (dimension){
30     case 0:
31         if (i < nbRows-1)
32             /* one move in dim 0 <=> i++ */
33             return anodeindex_Inc(aNodeIndex,0);
34     case 1:
35         if (j < nbCols-1)
36             /* one move in dim 1 <=> j++ */
37             return anodeindex_Inc(aNodeIndex,1);
38     }
39     return NULL;
40 }
41 /*-----*/
42 /* Create an Abstraction Graph for a matrix */
43 AGraph newMatrixAbstraction(int nbRows, int nbCols){
44     return agraph_new(2, getRoot, next);
45
46     /* with 2 : the number of dimension */
47     /* getRoot : the root function */
48     /* next : the access function */
49 }
50 /*-----*/

```

Figure 10. AGraph description

screen dumps here. They come from a Cholesky factorization of a sparse symmetric positive definite matrix. An IBM SP2 has been used to collect traces during the execution of this application.

We interfaced our tools with a data parallel system: the source program is written in a HPF2-like language and compiled using ADAPTOR[10] extended with a library called DDDT (Distributed Derived Data type for Tree). This library allows the manipulation of hierarchical access irregular data structures[7]. For an efficient parallel execution, a specific irregular distribution[11] is used.

9.3. Data Distribution Display

Figure 15 shows the distribution of the data on the virtual processors (one color per processor).

Using the mouse, the user can select a virtual processor (resp. piece of data) and visualize the corresponding data item (resp. processor). *Data Distribution Display* can be

used before the execution if the distribution is available at that time, i.e. if it is regular or computable.

9.4. Trace Data Display

Trace Data Display is a post mortem tool based on runtime generated traces and on the mapping defined by our model. The parallel application is instrumented so that its execution generates traces in terms of accesses to data items. At runtime, a trace collector process records events on each processor. Filtering methods are used to limit the amount of traces.

Figure 16 shows which data items are involved in remote read operations during the first part of the Cholesky factorization [11]. This is a user requirement that helps to estimate the quality of the data distribution.

10. Conclusion and future work

In this paper we have set up a formal approach for the modelization of sparse and irregular data structures. Using this framework, we have shown that it is possible to describe implementations, only introducing semantics in the mapping from the implementation to the abstraction. Depending on what they do, tools built on top of this framework can either use the abstraction graph, or, to be more efficient, use the mapping that makes it possible to scan the data structure avoiding to look at holes.

Another level of abstraction that we still have to implement is that of filtering. Assume that within our model, we are provided with an abstraction that represents a matrix as shown figure 9. The user may want to see either the matrix itself, or, for instance, a given row of this matrix (see figure 17), or even all items of each row as a unique item, i.e. he needs to filter the abstraction. Therefore we are working on the definition of a view graph with a mapping from the abstraction to it.

The next steps within this project are:

1. designing libraries that would provide standard abstractions using this model for standard implementations, like those of SPARSKIT for instance;
2. extending the high level tools based on this model, to add them to Visit[7] inside the HPFIT environment;
3. validating the tools with the end-users to propose views adapted to their needs;
4. interfacing this framework to existing software environments such as DAQV[18] or EMILY[17].

References

- [1] M. Alabau. *Une expression des algorithmes massivement parallèles à structures de données irrégulières*. Thèse, LaBRI — Université BORDEAUX I, Sept. 1994.
- [2] T. Bemmerl. An integrated and portable tool environment for parallel computers. In *Proceedings of the IEEE International Conference on Parallel Processing (St. Charles, USA)*, pages 50–53, 1988.
- [3] T. Bemmerl and A. Bode. An integrated environment for programming distributed memory multiprocessors. In B. A., editor, *Proceedings of the Second European Distributed Memory Computing Conference (München), Volume 487 of Lecture Notes in Comput. Sci.*, pages 130–142. Springer-Verlag, 1991.
- [4] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. Technical report, University of Leiden.
- [5] A. J. C. Bik and H. A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, (31):14–24, 1995.
- [6] A. Bode. Developments in distributed memory architectures. In *Proceedings of Microsystem '90 (Bratislava, CSSR)*, 1990. Also in Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung Paralleler Rechner Architekturen, TOPSYS, Tools for Parallel Systems, TUM-19013, SFB-Bericht Nr. 342/9/90 A, January 1990, seiten 11–16.
- [7] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J. Mignot, F. Desprez, and J. Roman. HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran: Part II: Data Structures Visualization and HPF Support for Irregular Data Structures with Hierarchical Scheme. *Parallel Computing*, 1996. Edited by J. Dongarra and B. Tourancheau.
- [8] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J. Mignot, F. Desprez, and J. Roman. HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran: Part I: HPFIT and the TransTOOL Environment. *Parallel Computing*, 1996. Edited by J. Dongarra and B. Tourancheau.
- [9] T. Brandes, S. Chaumette, and F. Desprez. TransTOOL: a tool for porting scientific applications on parallel distributed memory machines. In *2nd European School of Computer Science, Parallel Programming Environments For High Performance Computing*, 1996.
- [10] T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K. Decker and R. Rehmman, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, Apr. 1994.
- [11] P. Charrier, F. L., and J. Roman. Block data partition for parallel nested dissection. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*. Siam Editions, 1995.
- [12] S. Chaumette, F. Rubi, and J. Lepine. Internal report, LaBRI, Université Bordeaux-I, 1997. To be published.
- [13] D. Cheng and R. Hood. A portable debugger for parallel and distributed programs. In *Proc. of Supercomputing '94*, 1994.
- [14] M. Hao, A. Karp, M. Mackey, V. Singh, and J. Chien. On-the-fly visualization and debugging of parallel programs.
- [15] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.
- [16] G. Hurteau, V. Van Dongen, and G. Gao. Overview of EPPP - an Environment for Portable Parallel Programming. In *Proceedings of Supercomputing Symposium '94, Canada's Eighth Annual High Performance Computing Conference*, pages 119–127, Toronto, Ontario, June 1994. Also available as <ftp://ftp.crim.ca/apar/public/Papers/1994/SS94-EPPP.ps.gz>.
- [17] T. Loos and R. Bramley. EMILY: a visualization tool for large sparse matrices. Available as <http://www.cs.indiana.edu/scicomp/emily.html>.
- [18] A. Malony, M. Zosel, M. John, A. Karp, and D. Presberg. PTOOLS project proposal – Distributed Array Query and Visualization. Available as <http://www.cs.uoregon.edu/hacks/research/ptools-daqv/proposal>.
- [19] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):61–74, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.
- [20] J. Pazat. *Spring School on data Parallelism*, chapter Tools for High Performance Fortran: a Survey. Springer Verlag, 1996.
- [21] R. Pozo. Template Numerical Toolkit . Available as <http://math.nist.gov/>.
- [22] B. S., G. W.D, M. L.C., and S. B.F. *Modern Software Tools in Scientific Computing*, chapter Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries, pages 63–202. Birkhauser Press, 1997. Also available at <http://www.mcs.anl.gov/petsc/petsc.html>.
- [23] Y. Saad. *SPARSEKIT: A Basic Tools Kit for Sparse Matrix Computations*, June 1994. Version 2. Documentation.
- [24] Y. Saad and A. V. Malevsky. *P-SPARSLIB: a portable library of distributed memory sparse iterative solvers*, 1995. Documentation.
- [25] G. Strang and G. F. Fix. *An Analysis of the Finite Element Method*. Prentice Hall, 1973.
- [26] The PHAROS Team. The PHAROS project. Available as <http://www.vcpc.univie.ac.at/activities/projects/PHAROS/>.

```

1  /*****
2  MAPPING DEFINITION (with Compress Sparse Column)
3  *****/
4  /* CSC : IGraph contains 3 Vectors (V0, V1, V2) */
5  /* For column j :V1[ji] (for ji between V0[j], V0[j+1]-1) stores */
6  /* the line indices of non zero elements. */
7  /* V2[ji] is the value of matrix[V1[ji],j] */
8  /*-----*/
9  extern IDS ids; /* the irregular data structure */
10 /*****
11 MAPPING FROM AGRAPH TO IGRAPH
12 *****/
13 /* INodeIndex of mat[i,j] if exists. */
14 INodeIndex abs_to_imp(Mapping mapping, ANodeIndex aNodeIndex) {
15
16     IGraph igrph = mapping_getIGraph(mapping);
17     /* i : number of moves in dimension 0 (rows) */
18     int i = anodeindex_depth(aNodeIndex, 0);
19     /* j : number of moves in dimension 1 (columns)*/
20     int j = anodeindex_depth(aNodeIndex, 1);
21     int ji;
22
23     if (j >= ids->V0_size) return NULL; /* j is too large */
24     /* searching i between V0[j], V0[j+1]-1 */
25     for (ji=ids->V0[j]; ji<ids->V0[j+1];ji++)
26         if (i == ids->V1[ji]) /* mat [i,j] is not a hole */
27             return inodeindex_new(igrph,idsindex_new(-1,-1,ji));
28     else if (i < ids->V1[k]) return NULL;
29     return NULL;
30 }
31 /*-----*/
32 ANodeIndex imp_to_abs(Mapping mapping,INodeIndex iNodeIndex) {
33     /* ANodeIndex of an IdsIndex with value (-1,-1,d2) */
34     /* searching [i,j] where i = V1[d2] and j / d2 >= V0[j] and d2 < V0[j+1]*/
35     ANodeIndex aNodeIndex;
36     IGraph igrph = mapping_getIGraph(mapping);
37     IDSIndex idsIndex = inodeindex_getUserData(iNodeIndex);
38     int d2 = idsIndex->d2;
39     int j;
40
41     if (d2 == -1) return NULL; /* No correspondent node */
42
43     /* searching column number */
44     for (j=0;j<ids->V0_size-1;j++)
45         if ((d2 >= ids->V0[j]) && (d2 < ids->V0[j+1])) break;
46     if (j==ids->V0_size-1) return NULL; /*d2 is too large */
47
48     /* agraph root index */
49     aNodeIndex = anodeindex_new(NULL,NULL,NULL);
50     /* do V1[d2] moves in dimension 0 */
51     anodeindex_multInc(aNodeIndex,0,ids->V1[d2]);
52     /* do j moves in dimension 1 */
53     anodeindex_multInc(aNodeIndex,1,j);
54
55     return aNodeIndex;
56 }
57 /*-----*/

```

Figure 11. Mapping description

```

1 void
2 agraph_MatrixOperation(AGraph agraph,
3 void (*matrixOperation)(int i, int j, void *value))
4 {
5     ANodeIndex i0, ij;
6
7     for (i0=agrph_getRoot(aGraph);i0!=NULL;
8         i0=anodeindex_next(0, i0))
9         for (ij=anodeindex_clone(i0);ij!=NULL;
10            ij=anodeindex_next(1, ij))
11             matrixOperation(anodeindex_depth(0, ij),
12                             anodeindex_depth(1, ij),
13                             anode_getValue(ij));
14 }

```

Figure 12. Scanning the data structure through the abstraction graph

```

1 void
2 agraph_MatrixOperation(AGraph agraph,
3 void (*matrixOperation)(int i, int j, void *value))
4 {
5     int i,j;
6
7     for (i=0; i<=n; i++)
8         for (j=0; j<n; j++)
9             matrixOperation(i, j, M[i,j]);
10 }

```

Figure 13. Matrix equivalent

```

1 void
2 mgraph_MatrixOperation(MGraph mgraph,
3 void (*matrixOperation)(int i, int j, void *value))
4 {
5     MNodeIndex i;
6
7     for (i=mgraph_getRoot(mGraph);i!=NULL;
8         i=mnodeindex_next(0, i)){
9         ANodeIndex a=mnodeindex_getANodeIndex(i);
10        matrixOperation(anodeindex_depth(0, a),
11                        anodeindex_depth(1, a),
12                        anode_getValue(a));
13    }
14 }

```

Figure 14. Scanning the data structure through the mapping

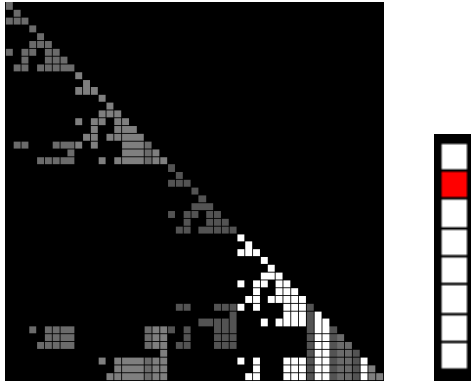


Figure 15. Data Distribution

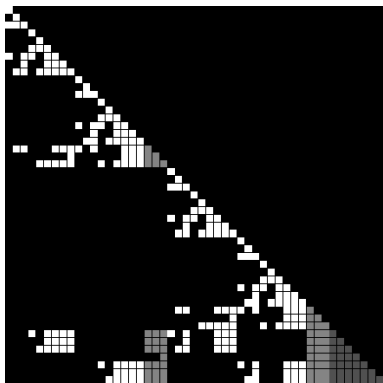


Figure 16. Trace Data



Figure 17. View as a row