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

Serge Chaumette, Jean-Michel Lepine, Franck Rubi

A graph based framework for the definition of tools dealing with sparse and irregular distributed data-structures[1]

Serge Chaumette, Jean-Michel Lépine and Franck Rubi

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

Tel.: (+33 5) 56 84 69 04 – Fax: (+33 5) 56 84 66 69

LaBRI, Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux I, 351 Cours de la Libération, 33405 Talence, France.

Keywords

Programming Environments, Data Parallelism, Tools, Visualization, Sparse Data Structures, Distributed Data Structures, Irregular Data Structures, Graphs

Abstract

Industrial applications mainly 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[21]. Specific implementations are especially oftenly used when dealing with large sparse and irregular data-structures, such as matrices coming from the domain of finite elements[23]. 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.

In this paper, we present a framework which we have setup to support the development of such tools, and prototypes tools based on it 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

[1]This works is supported by the French GDR-PRC PRS.

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 effort[2] *[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. The aim of this project is to provide an integrated HPF development environment that supports sparse and irregular data structures.*

1 Introduction and 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, 5]) or to develop dedicated libraries, to provide programmers with ready implementations of possibly sparse distributed data structures (see for instance SPARSEKIT[21], P-SPARSLIB[22] or PETSc[20]).

Tools exist, but they mainly deal with **regular** (HPF-like) data structures (see for instance EPPP[16], P2D2[13], Pharos[25], Paradyn[18]). 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 [19] 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 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 either as fully supported or simply as ported tools. 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).

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

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 proceed 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 [17]). This is not straightforwardly portable.

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.

The rest of this paper is organized as follows. We first describe the overall architecture of the model we propose in section 2. Section 3 introduces the way we modelize data structures. We then describe the three current levels of our model in sections 4, 5 and 6. Section 7 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.

2 General architecture of the model

Our model is composed of four levels. Each of these levels, i.e. implementation, abstraction, mapping and view is a graph (figure 2):

1. IGraph: the Implementation Graph (figure 1(a)) 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 1(b)) describes the abstract data structure the application developer has in mind – e.g. a matrix.

3. MGraph: the Mapping Graph describes the relationship between the IGraph and the AGraph. This graph is a bridge that carries the semantics between the implementation and the abstraction.

4. VGraph: the View Graph (figures 1(c) and 1(d)) 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 8), but one can work with the AGraph, which is equivalent to having an identity between the AGraph and the VGraph.

Although this will not be detailed here, the model provides the same efficiency when working at the mapping graph level or at the implementation graph level (see figure 3).

3 Modelization of a data structure

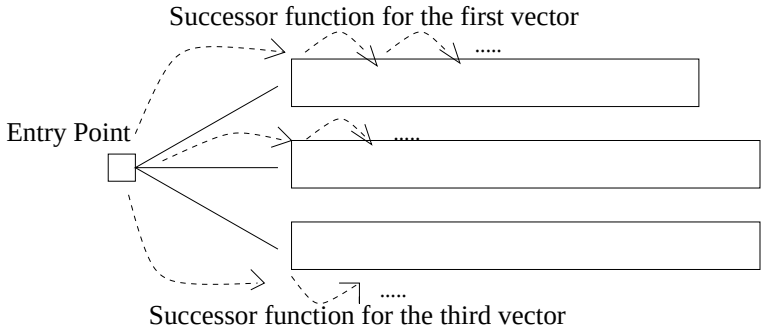
We define a data structure in terms of entry points and access functions. This approach reflects the way a data structure is implemented. It leads to a modelization in terms of a graph. For instance a vector `int v[10]` can be represented by a graph, the root of which is `v[0]` and the nodes are `v[i]`.

Definition 1 *Data structure*

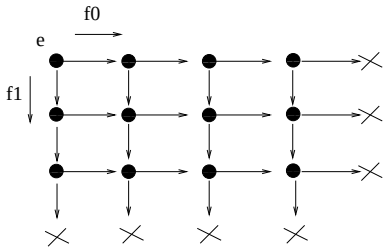
A data structure DS is a set of **data items** *that are* **structured** *by means of* **access functions**.

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

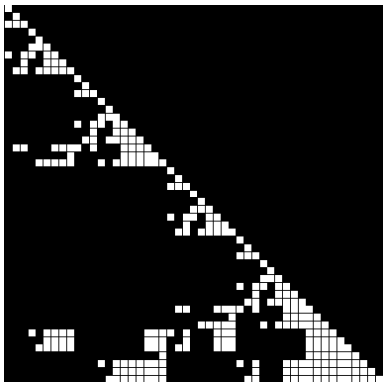
where :



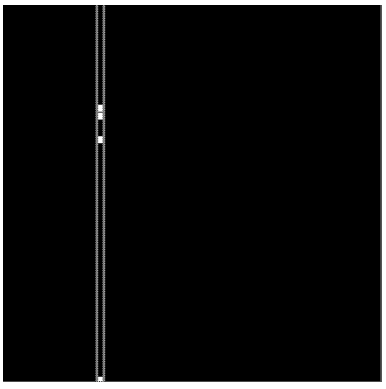
(a) Implementation



(b) Abstraction as a matrix



(c) View as a matrix



(d) View as a column

Figure 1: Architecture of the framework

- *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.*

Irregular data structures can be described using this framework: 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

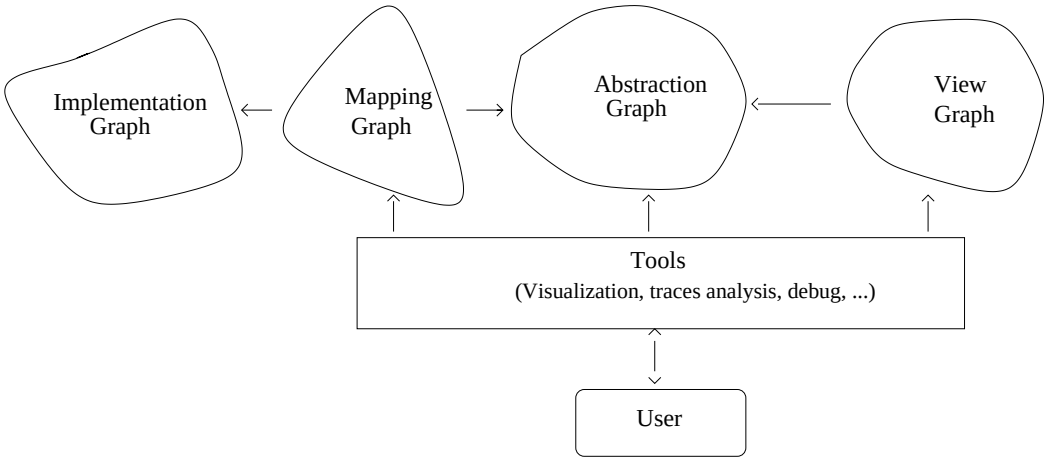


Figure 2: Architecture of the model

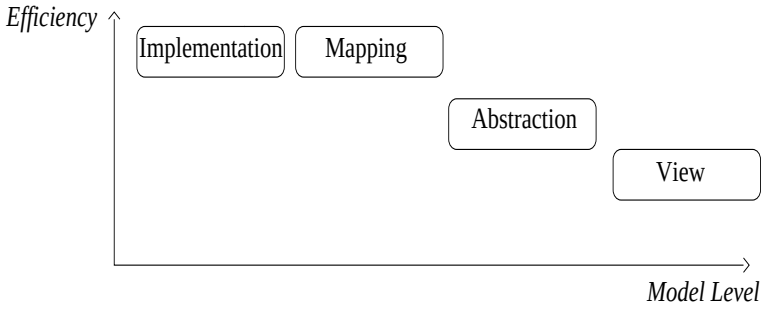


Figure 3: Efficiency of the model levels

data-structure.

Definition 2 *Access function*

An **access function** *is a function over the nodes of the data structures that being given a node produces another node.*

$$\begin{array}{rccc} f: & N & \longrightarrow & N \\ & n & \longmapsto & f(n) \end{array}$$

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

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

Definition 3 *A data structure is a graph*

$$DS = (N, d, F, r, R)$$

where :

- *N is a set of nodes;*
- *d is the number of successor functions;*
- *F is a d-uple of successor functions;*
- *r is the number of roots of the graph.*
- *R is the set of roots of the graph.*

There are some data structures, where each data item can be accessed directly, like sets; we call them direct access data structures.

Definition 4 *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.

4 Describing the implementation data structure

In this section we present an example, the aim of which is to illustrate how the definitions of section 3 can be used to model an effective implementation.

The implementation contains three vectors of different sizes. 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.

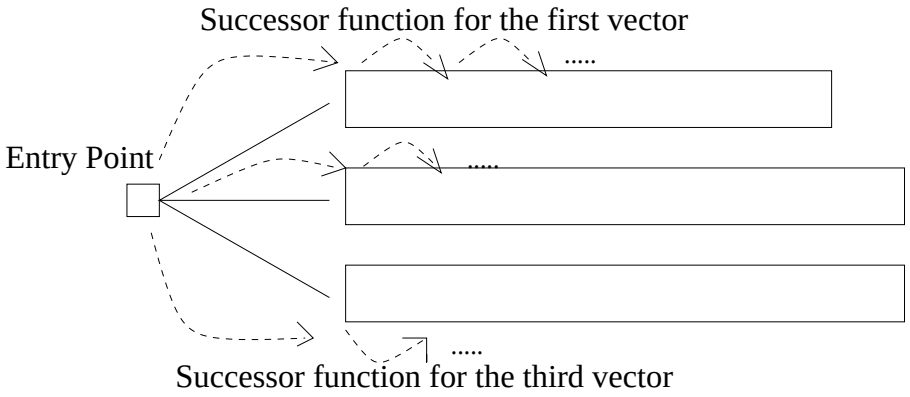


Figure 4: A graph modelization of the irregular structure

Figure 5 shows a sample of the code the programmer must write in order to describe this implementation.

5 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 4 can be interpreted as a CSC storage of a matrix (see figure 6).

Hence, abstracting from an implementation consists in mapping the implementation graph to the abstraction graph.

6 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 graph, the nodes of which are defined as a pair 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 than the implementation which is sparse). Assume we have 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 5 $Nodes = \{(abs\_to\_imp(anode), anode) / anode \in \text{agraphe nodes}\}$

Remark We can also define a function *imp_ to_ abs* (implementation node to abstraction node). It allows another symmetric definition of the mapping graph. The user can define the two functions (to gain efficiency). One of these functions may be difficult to write: it can be left to the tools to construct it using the other function, storing this information in a hash table.

7 Usage

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

7.1 General principle

Using our libraries, high level tools work only with graphs, which they can for instance scan, using either of the loops shown figures 7 and 8.

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.

7.2 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 graphs 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 screen dumps here. They come from a Cholesky factorization of a sparse symmetric positive definite matrix. The parallel machine which has been used to run it to collect traces is an IBM SP2.

We interfaced our tools with a data parallel system. The source program is written in a HPF2-like language and compiled using ADAPTOR[10] extented 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.

7.3 Data Distribution Display

Figure 9 shows the distribution of the data on the virtual processors (one color by processor).

Using the mouse, the user may select a virtual processor (resp. data) and visualize the corresponding data item (resp. processor). Data Distribution Display can be used before the execution if the distribution is regular or computable.

7.4 Trace Data Display

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

Figure 10 shows which data items are involved in remote read operations during the first part of the Cholesky factorization [11]. This can for instance help the user to estimate the quality of the data distribution.

8 Conclusion and future work

In this paper we have set up a formal approach to 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 of what they are doing,

tools build on top of this framework can either use the abstraction graph, or, to be more efficient, use the mapping graph 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 11(a). The user may want to see either the matrix itself, or, for instance, a given row of this matrix (see figure 11(b)), 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[17] or EMILY[24].

References

[1] M. Alabau. *Une expression des algorithmes massivement parallèles à structures de données irrégulières*. Thèse, LaBRI — Université BORDEAUX I, September 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 Bode 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-I9013, SFB-Bericht Nr. 342/9/90 A, January 1990, seiten 11–16.

[7] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. 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.C. 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.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, April 1994.

[11] P. Charrier, Fack 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 Lepine J.M. Internal report, LaBRI, Université Bordeaux-I, 1997. To be published.

[13] Doreen Cheng and Robert Hood. A portable debugger for parallel and distributed programs. In *Proc. of Supercomputing'94*, 1994.

[14] M.C. Hao, A.H. Karp, M. Mackey, V. Singh, and J. Chien. On-the-fly visualization and debugging of parallel programs.

[15] M.T. Heath and J.A. 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] A. Malony, Mary Zosel, May John, Alan Karp, and David Presberg. PTOOLS project proposal – Distributed Array Query and Visualization. Available as http://www.cs.uoregon.edu/ hacks/research/ptools-daqv/proposal.

[18] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia 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.

[19] J.L Pazat. *Spring School on data Parallelism*, chapter Tools for High Performance Fortran: a Survey. Springer Verlag, 1996.

[20] Balay S., GroppD W.D, McInnes L.C., and Smith 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.

[21] Y. Saad. *SPARSEKIT: A Basic Tools Kit for Sparse Matrix Computations*, June 1994. Version 2. Documentation.

[22] Y. Saad and A. V. Malevsky. *P-SPARSLIB: a portable library of distributed memory sparse iterative solvers*, 1995. Documentation.

[23] G. Strang and G. F. Fix. *An Analysis of the Finite Element Method.* Prentice Hall, 1973.

[24] Loos T. and Bramley R. EMILY: a visualization tool for large sparse matrices. Available as http://www.cs.indiana.edu/scicomp/emily.html.

[25] The PHAROS Team. The PHAROS project. Available as http://www.vcpc.univie.ac.at/activities/projects/PHAROS/.

```
/*****************************************************************************
  IMPLEMENTATION NODE INDEX DEFINITION
*****************************************************************************/
extern IDS iDS; /* The Irregular Data Structure */
/* composed here of 3 Vectors of different sizes */
/*---------------------------------------------------------------------------*/
struct IDSIndex{
  int d0;    /* an index for the Irregular Data Structure, a value          */
  int d1;    /* di (i=0, 1 or 2) give a position in the corresponding vector */
  int d2;
};
/*---------------------------------------------------------------------------*/
/* function used to create a new IDSIndex                                    */
IDSIndex idsindex_new(int v0, int v1, int v2){
 ... allocate the idsindex and copy v0, v1, v2 to do, d1, d2 ...
 return iDSIndex;
}
/*****************************************************************************
  IMPLEMENTATION GRAPH DEFINITION (IGraph)
*****************************************************************************/
/* function used to give the index root                                      */
static IDSIndex getRoot(IGraph iGraph){
 return idsindex_new(-1,-1,-1));
}
/*---------------------------------------------------------------------------*/
/* this function returns the next node if exists in dimension dim            */
static IDSIndex next( int dim, IDSIndex iDSIndex){
  /* we allow only move in one vector at a time                       */
  /*  => only one di must be different of -1                          */
  /*  at the end of a vector, we return NULL value                    */
  switch (dim) {
    case 0:
      if ((iDSIndex->d1 != -1) || (iDSIndex->d2 != -1)) return NULL;
      if (iDSIndex->d0 < iDS->V0_size-1)
        return idsindex_new(iDSIndex->d0+1,iDSIndex->d1, iDSIndex->d2));
      break;
    case 1:  ...  the same as case 0 but for dimension 1 or 2 ...
    case 2:
    }
  return NULL;
}
/*---------------------------------------------------------------------------*/
/* this function returns for an iDSindex the corresponding value             */
static void * getNodeValue( IDSIndex iDSindex){
  /* only one di (i = 0,1 or 2) has a value different from -1 */
  if (iDSIndex->d0 != -1) return (void *) &(iDS->V0[iDSIndex->d0]);
  if (iDSIndex->d1 != -1) return (void *) &(iDS->V1[iDSIndex->d1]);
  if (iDSIndex->d2 != -1) return (void *) &(iDS->V2[iDSIndex->d2]);
  return NULL;
}
/*---------------------------------------------------------------------------*/
```

Figure 5: IGraph description

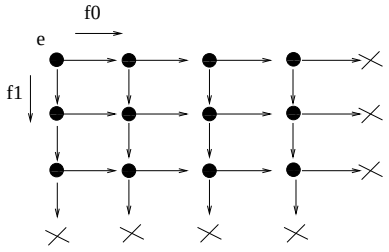


Figure 6: Abstraction as a dense matrix graph

```
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=agraph_getRoot(aGraph); i0!=NULL; i0=anodeindex_next(0, i0))
8       for (ij=anodeindex_clone(i0); ij!=NULL; ij=anodeindex_next(1, ij))
9         matrixOperation(anodeindex_depth(0, ij),
10                        anodeindex_depth(1, ij),
11                        anode_getValue(ij));
12  }
```

Figure 7: Scanning the data structure through the abstraction graph

```
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; i=mnodeindex_next(0, i)){
8       ANodeIndex a=mnodeindex_getANodeIndex(i);
9       matrixOperation(anodeindex_depth(0, a),
10                      anodeindex_depth(1, a),
11                      anode_getValue(a));
12    }
13  }
```

Figure 8: Scanning the data structure through the mapping graph

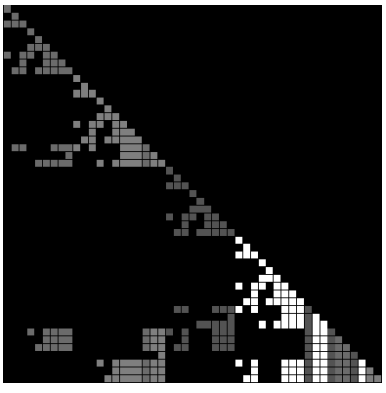


Figure 9: Data Distribution

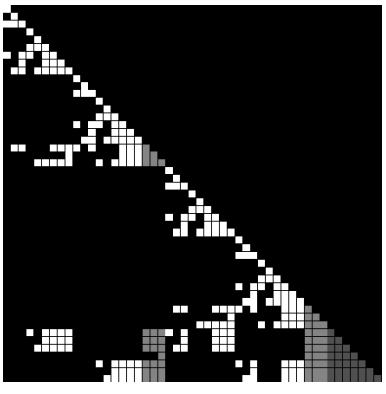
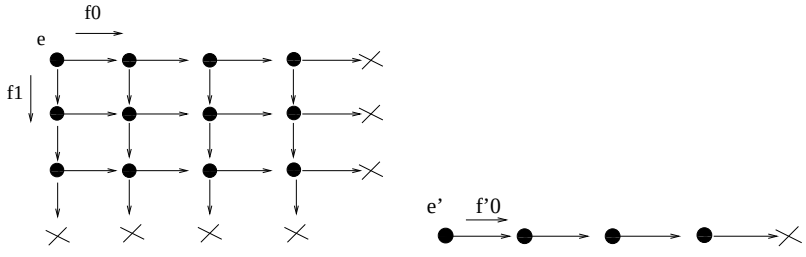


Figure 10: Trace Data



(a) Abstraction as a matrix (b) View as a row

Figure 11: Introduction of a view graph