Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran

T. Brandes, S. Chaumette, M.C. Counilh, A. Darte, F. Desprez, J.C. Mignot, J. Roman

October 8, 1996

Research Report N^{o} 96-28

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran

T. Brandes, S. Chaumette, M.C. Counilh, A. Darte, F. Desprez, J.C. Mignot, J. Roman

October 8, 1996

Abstract

In this report, we present the HPFIT project whose aim is to provide a set of interactive tools integrated in a single environment to help users to parallelize scientific applications to be run on Distributed Memory Parallel Computers. HPFIT is built around a restructuring tool called TransTOOL which includes an editor, a parser, a dependence analysis tool and an optimization kernel. Moreover, we provide the users with a clean interface, so that developers of tools around High Performance Fortran can easily integrate their software within our tool.

Keywords: Semi-automatic parallelization, High Performance Fortran (HPF), Development Tools

Résumé

Dans ce rapport, nous présentons le projet HPFIT dont le but est de fournir un ensemble d'outils interactifs intégrés dans un seul environnement pour aider les utilisateurs à paralléliser des applications scientifiques sur des machines parallèles à mémoire distribuée. HPFIT est bâti autour d'un outil de restructuration appelé TransTOOL qui inclut un éditeur, un analyseur syntaxique, un outil d'analyse de dépendances et un noyau d'optimisation. De plus, nous fournissons une interface propre pour aider les développeurs d'outils autour d'High Performance Fortran à intégrer leurs logiciels à l'intérieur de notre outil.

Mots-clés: Parallélisation semi-automatique, High Performance Fortran, outils de développement

HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran[1]

T. Brandes
GMD/SCAI[2]
Germany

S. Chaumette, M.C. Counilh and J. Roman
LaBRI[3]
France

A. Darte and F. Desprez and J.C. Mignot
LIP[4]
France

October 8, 1996

[1]This work is partly supported by the CNRS–ENS Lyon–INRIA project *ReMaP* and by the *LHPC–EuroTOPS* project.

[2]Institute for Algorithms and Scientific Computing, German National Research Center for Computer Science, Schloss Birlinghoven, PO Box 1319, 53754 St. Augustin, Germany.
[3]URA CNRS 1304, ENSERB and Université Bordeaux I, 351, Cours de la Liberation, 33405 Talence, France.
[4]URA CNRS 1398, INRIA Rhône-Alpes, 46, Allée D'Italie, 69364 Lyon Cedex 07, France.

Introduction

Developing large scientific applications on distributed memory parallel computers is a difficult task, especially for newcomers or people without a deep knowledge of parallelism. Currently, we notice that parallelism not only enters industrial areas but also other fields of science. The aim of European projects, like EUROPORT 1 and 2 is clear: to prove the validity of a parallel solution for large industrial codes. Moreover, in order to solve larger problems, scientists need more and more powerful computers in terms of Mflops and memory size. During the last two years, big efforts have been put in the definition of libraries and languages. These efforts lead to standards like BLAS (Basic Linear Algebra Subroutines), MPI (Message Passing Interface) and HPF (High Performance Fortran). Some parallel libraries like ScaLAPACK or NAG start to be widely used. Finally, many tools for the development of applications using data-parallel languages begin to appear. Nevertheless, almost none of them offers a complete collection of portable tools and almost none of them is available as a free software.

It is unlikely that we will ever see a "black box" able to parallelize a non-trivial serial code into a performing parallel code. The user needs to help the compiler by giving information about his code. This can be done for example via directives inserted in the source file like in HPF. But it is now admitted that the insertion of such directives is a non trivial task for average users who do not have a deep knowledge of parallelization techniques. Therefore, interactive parallelization tools have a great importance in parallel computing, even in the limited field of numerical problems.

This report includes two papers, presenting the HPFIT [1] project and the developments made around it. These two papers were presented at the ETPSCIII workshop [14, 15]. The first paper presents the HPFIT project in general and its kernel, TransTOOL (Chapter 1). The second paper presents the data structure visualization tool VisIt and HPF extensions for irregular problems (Chapter 2). These two papers describe the first results of our development effort.

The remainder of this report is organized as follows. In Chapter 1, Section 1.1 gives a short survey of tools for data-parallel programming. Section 1.2 presents the development of a parallel application and describes the HPFIT project. In Section 1.3, we present TransTOOL which contains the editor, the parser, the dependences analysis tool, and an optimization kernel. Section 1.4 presents two research directions in the TransTOOL optimization kernel. In Chapter 2, Section 2.1 describes one tool, DDD, used to visualize and analyse data-parallel programs in terms of the data they work on, and its underlying model. Section 2.2 presents extensions of High Performance Fortran to handle irregular problems before some conclusions and ideas for future work.

[1] URL `http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/SOFT/HPFIT/`

Chapter 1

HPFIT and the TransTOOL Environment

1.1 Previous Work

A lot of work has been done since the early eighties around tools for supercomputer programming. These early tools need to be enhanced to follow the development of parallel computing. In this section, we present some tools for the parallelization of applications written in Fortran 77 and HPF. For a comprehensive survey of HPF tools, see [48].

One of the first projects around an "intelligent" editor for the parallelization of applications written in Fortran77 was the ParaScope editor (PED) from Rice University [42]. PED was designed for shared memory machines. It has been built from different other projects in Rice, like $\mathbb{R}_n$, PFC and PTOOL. PED allowed the search of a dependences graph whose size was limited by some filtering information. Several optimizations were added to the tool like loop restructuring, loop parallelization, dependence deletion and memory optimization. PED has been using an incremental analysis to update its text and dependences panes.

The D-Editor [35] has been partly developed from PED also at Rice University. This editor has been designed for the parallelization of applications written in Fortran D, a data-parallel language which turned out to be a major input to the design of HPF. The D-editor contains an interprocedural analysis tool, the Fortran D compiler and tools for automatic data distribution, data-race detection, static performance estimation and performance profiling. The graphical display is derived from PED and contains five panes: an overview pane provides a summary of the loops and subroutines in the program (loops which restrict parallelism are highlighted); a dependence pane displays the data dependences carried on the selected loop; the communication pane displays all the communications associated with the selected loop; the data layout pane displays the data decomposition information for each array of the loop; and finally the source pane shows the actual program code. The performance analysis environment Pablo has also been integrated within the D-Editor [2].

The Vienna Fortran Compilation System (VFCS) [60] is a source-to-source compilation system based on Vienna Fortran, another extension set for Fortran, similar to Fortran D and HPF. It contains a compiler, an interactive performance estimator P3T [20], a performance measuring system (VFPMS), and a knowledge based tool, eXPert Advisor (XPA) to help the user to insert the directives [5].

ForgeExplorer is an interactive parallelizer based on an interactive source code browser. It also performs loop level transformations.

The Annai tool environment [22], developed by CSCS in a collaboration with NEC, uses MPI as the communication interface for various distributed memory platforms: NEC Cenju-3, Cray T3D, Intel Paragon, and Unix multiprocessor/networked workstations. It consists of an extended HPF compiler (with extensions for irregularly structured computations), a parallel debugger and performance monitor and analyzer, designed with important feedback from application developers.

The Computer Aided Parallelization Tools (CAPTools [37]) is a recent set of tools developed at the

University of Greenwich. This tool can show dependences using a graphical display [44]. The user can interact inside the parallelization process by giving information about values or ranges of variables, by "deleting" dependences, ...It has a Partitioner window to partition his code and data structures, a communication browser to see the generated communication routines calls. The code generated by CAPTools is written in F77 and explicit communications routines calls.

To conclude this section, we point out that other important compilation projects are being undertaken, like SUIF [4] and Paradigm [7].

1.2 The HPFIT Project

The development of a "real" application is conducted in a cycle involving several steps. This cycle is shown on Figure 1.1.

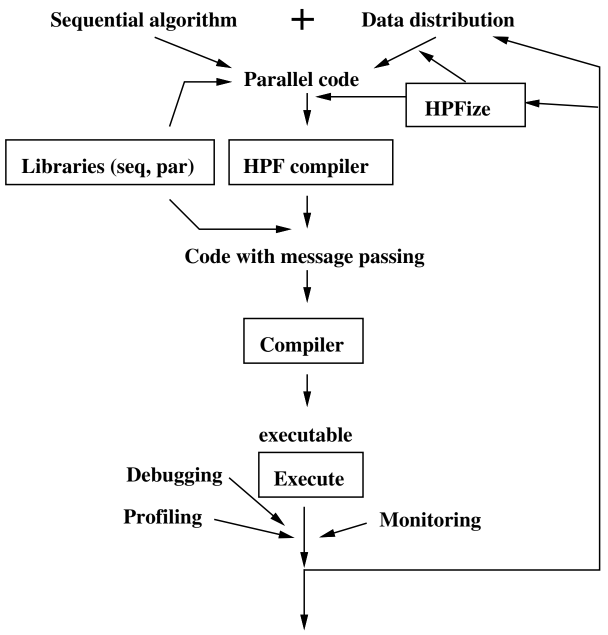

Figure 1.1: Development cycle of an application on distributed memory platforms.

Tools can be designed to help users especially during the distribution phase. The programmer must find a "good" distribution of data structures. This distribution should allow for the most parallelism, and should reduce the number and volume of communications to its minimum. This can be evaluated using various tools (monitoring tools, profilers). From this distribution, the user is able to write HPF directives in the declaration part of the code. These distributions should be propagated inside the routines. They can sometimes differ and the programmer is allowed to redistribute the data inside the code. The code is then compiled to obtain a source code in Fortran 77 with message-passing calls. It is sometimes possible to modify the resulting code to insert optimizations. The code can then be executed or simulated. If traces have been generated, the user can have an idea of the behavior of his code. The quality of the distribution (and the optimizations) can be improved. This cycle can be executed several times to obtain the best performance. One question is: how portable is the resulting code?

One problem with HPF is that a HPF compiler is not required to follow the user's advice (stated as directives). This can be a problem because the user can have a false view of his code, "corrupted" by the compiler. That is why we think that a strong interaction between the compiler and the parallelization tool is necessary.

The aim of the **HPFIT**[1] project is to provide a set of interactive tools integrated in a single environment to help users to parallelize scientific applications to be run on distributed memory platforms. This tool will also be used as an interface to a large number of existing tools like HPF compilers, computation libraries, simulators, data visualization tools, monitoring systems, profilers, and so on. These tools will interact in a coherent environment. This environment should allow scientists with sequential source codes to produce good parallel versions. Furthermore HPFIT will enable users to control the parallelization of their application, in order to make it even better.

HPFIT will not be a black box taking a sequential application as input and magically producing an efficient parallel application as output. Rather, HPFIT will be a tool environment to support and ease the development, tuning and maintenance of HPF applications. Though it is intended to be an open environment that allows integration of new tools, we focus in the first version of our tool on the following items:

- source editing with analysis information,
- dependence analysis of particular loop nests,
- automatic detection of `independent` loops,
- automatic detection of pipelined computations patterns and code generation,
- support for data mapping (adding data distribution directives),
- visualization and evaluation of data mappings,
- interfaces to existing parallel libraries (e.g. ScaLAPACK, ...),
- interfaces to HPF compilers,
- simulation, monitoring, performance analysis and interface with profiling tools,
- compilation and execution interface.

The parallelized code is a data parallel program with explicit data mapping and explicit data parallelism. As the data parallel paradigm might not be the most efficient one in some situations, we will provide a library interface to codes written in other languages or other parallel programming styles, in particular message passing libraries, and efficient parallel computation libraries.

As most applications considered for parallelization are written in Fortran, and because a lot of work has been done for automatic parallelization of this language, our target language is HPF. Since HPF provides the `EXTRINSIC` mechanism, we can interface it with existing parallel libraries and other programming styles. Some HPF compilers are being released. These compilers are designed by software companies (like pghpf from PGI, DEC and IBM HPF compilers, ...) or universities (like ADAPTOR from GMD/SCAI [19], VFCS from the University of Vienna, sHPF from the University of Southampton, Fortran D from Rice University, PARADIGM from the Center for Reliable and High-Performance Computing at the University of Illinois at Urbana-Champaign, HPFC from the Ecole des Mines de Paris, ...). HPFIT should make possible for the user to use any of these HPF or HPF-like compiler, and all those to come, even if most of them will not allow some functionalities like monitoring, and translation of sequential library calls. We intend to make use of this variety of possible "post-compilers" in order to run HPF codes on nearly all kinds of architectures. For instance the ADAPTOR compilation system not only generates message passing code for MIMD system with distributed memory, but can also generate code for MIMD systems with shared or distributed shared memory. The first version of HPFIT (Version 1.0) will support two compilers: ADAPTOR from GMD/SCAI and pghpf from Portland Group.

The HPFIT project is based on several other projects developed in different universities. At the moment, 4 research groups have decided to work on the project and to design shared interfaces. These laboratories are the **LIP** in Lyon, France, the **LaBRI** in Bordeaux, France, the **LIFL** in Lille, France, and the **GMD/SCAI** in Bonn, Germany. The developed tools can be used either in a stand-alone fashion or within the HPFIT interface.

[1] **H**igh **P**erformance **F**ortran **I**ntegrated **T**ools.

1.3 TransTOOL

TransTOOL, developed at the LIP, is the kernel of the HPFIT project. At the moment, it contains a powerful editor (XEmacs), the F77 parser (from the **ADAPTOR** compiler developed at the GMD/SCAI lab.), the dependence analyzer (**Petit** at the University of Maryland) and an optimization kernel. At the moment, this kernel allows to do some parallelism detection and optimizations of pipelined computations (see Section 1.4). TransTOOL provides an interface to be able to get the results of the parsing and of the dependence analysis. Figure 1.2 gives a snapshot of the TransTOOL screen.

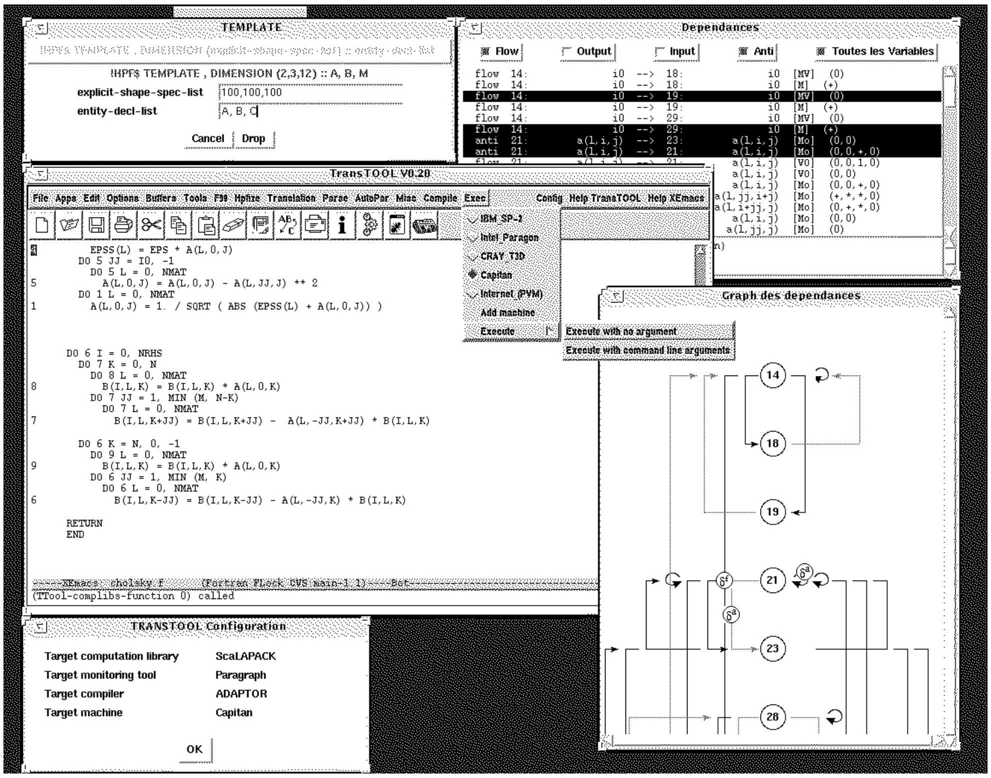

Figure 1.2: Snapshot of TransTOOL.

1.3.1 The XEmacs Editor

The sequential program source is displayed by an XEmacs editor. Using a powerful editor enables us to share previously developed modes, to let the user configure its editor with his preferences (by using his regular .emacs file). The TransTOOL edition is suited to the target language using a modified F90 mode. HPF keywords are highlighted. The user has the opportunity to click on a program component to trigger some actions (like choosing a loop nest for the dependence analysis).

1.3.2 HPFize

Numerous applications have been developed on sequential, vector or parallel machines. These codes have to be modified to be executed on distributed memory machines. Writing a program in HPF consists in including compilation directives into the source code. It is interesting to simplify the way the user inserts such directives. This is achieved using a graphical environment that is able to get the necessary pieces of information from the user and from the program itself (after parsing and dependence analysis).

To summarize, the first functionality of this part of TransTOOL is to help the user to insert HPF basic components into his "old Fortran" source code (and give defaults values as much as possible): let the user insert, with some assistance, directives and constructions like template, processor, align, distribute, forall and calls to intrinsic procedures.

The main research topic around the semi-automatic insertion of HPF directives is the automatic distribution of matrices using dependence analysis, previous distributions and target machine parameters.

1.3.3 Dependence Analysis

Dependence analysis is a crucial part of the semi-automatic parallelization of a code. Many tools for dependence analysis have been designed and Petit [40] is one of them. Petit is a version of Michael Wolfe's Tiny tool extended by the Omega Project at the University of Maryland. This tool uses the Omega library [41] to compute the dependences.

We use Petit via its batch interface to compute the dependences of selected loop nests. The user can choose a loop nest and ask for the dependences. Then, a graphical interface allows the user to see the dependences, to select some dependences according to some criterion (for example, choosing only the flow dependences, ...), to see the sink and target of dependences on the editor.

When a loop nest is chosen, the corresponding sub-program is rebuilt from the Abstract Syntax Tree and transformed into the Petit language using f2p.

Figure 1.2 shows some of the dependences analysis windows of TransTOOL.

1.3.4 Parsing and Unparsing

ADAPTOR (Automatic Data Parallelism Translator) is a public domain compilation system developed at GMD for compiling data parallel HPF programs to equivalent message passing programs [19]. The compiler tools used for ADAPTOR can be retrieved and used to build other tools. In TransTOOL, we use the front end which is able to parse a Fortran 77 source file, to generate the Abstract Syntax Tree (AST) and to unparse the AST in a output file. We use the AST and the unparsing functionality to build the sub-programs for the dependence analysis. If the source file is an HPF source code, the directives are also in the AST. We will use them for the semi-automatic parallelization.

We also use ADAPTOR for the compilation of the code generated by HPFIT.

1.3.5 Translation of Calls to Sequential Libraries

Many real applications are currently using basic libraries like BLAS or LAPACK. These libraries are available on many existing machines. Parallel versions of these libraries are available like the PBLAS [39] and ScaLAPACK [38]. They are highly optimized, and reach very high efficiencies close to peak performance together with a good scalability. It is impossible to reach the same efficiency using usual compilers. A problem appears when one wants to transform a source code into HPF: these libraries are added during the link phase, and thus their source code is not available for automatic parallelization during the upstream operations. Moreover, even if we have the source of the sequential routine, its automatic parallelization will lead to poor performance. If a parallel version of the library exists, we must use it for the parallelization of the application.

Some work have been done around HPF interfaces of parallel libraries [18, 45]. This is the best way to obtain a portability between HPF compilers and parallel libraries. One of the goals of TransTOOL is to offer the users the opportunity to automatically translate library calls from sequential to parallel versions (via their HPF interfaces). Furthermore, TransTOOL will allow the user to insert redistribution phases if it appears to be necessary. It will be a semi-automatic translation linked with the compilation. Moreover, TransTOOL will allow to insert the source code of subroutines which do not have their parallel implementation. Then the source will be "HPFized" and distribution will be inherited. One problem is that HPF handles many more distributions than those supported by ScaLAPACK. This will imply the development of conversion routines.

1.3.6 Execution Interface

TransTOOL has been designed to be a self-contained environment. From the XEmacs editor, the user should be able, taking a sequential Fortran 77 code, to semi-automatically generate an HPF source code, to compile this code and to execute the SPMD program on the different machines he has access to. To this purpose, we have designed an interface to give the parameters of the machines (how to start a computation on the machine, how you allocate nodes, and so on), to compile the HPF code by choosing among available HPF compilers, and to start the program on a remote machine.

1.3.7 Developer's Toolkit

HPFIT will provide a standard interface to many other tools used in the parallelization of applications like performance monitors, trace analyzers, and simulation tools. In this first version, HPFIT is not a totally new environment built from scratch but an "intelligent" interface into which existing or new tools are being plugged. Thanks to this interfacing, we will be able to add new tools as they appear.

The TransTOOL Developer's Toolkit (T^3) is the set of interfaces which can be used to build new tools from TransTOOL, or to integrate new functionalities inside the editor. Currently, the Toolkit has interfaces to:

- the XEmacs editor,
- the parser,
- the dependence analysis tool.

These interfaces are written with either C, TCL or Lisp, so they can be used in a C program, a tcl script or within the XEmacs editor.

The first version (V 1.0) of TransTOOL and its developer's Toolkit is available on the Web[2].

1.3.8 Other Tools

HPFbuilder [25] from the LIFL which allows the user to insert HPF distribution directives using a graphical interface will be integrated soon.

1.4 TransTOOL Optimization Kernel

Our main interest in TransTOOL is to validate recent research results on real applications, and to be able to integrate the corresponding (limited size) software developments within existing, more complete and more powerful tools.

In this section, we present two research topics, i.e. parallelism detection for automatic generation of `independent` directives, and the pipelined loops detection for the automatic generation of calls to the LOCCS library.

1.4.1 Parallelism Detection in Nested Loops

One of the objectives of the TransTOOL project is to develop and integrate strategies for transforming automatically (or semi-automatically) sequential Fortran pieces of codes into codes with HPF directives. The goal is to help the programmer to recognize parallelism at the loop level, and to automate the corresponding loops transformations for him.

Since our target language is HPF, we have to keep in mind that only transformations that can be expressed in HPF and that can be efficiently compiled by an HPF compiler are suitable. In particular, we do not currently address the following topics: parallelism exploited in doacross loops, software pipelining, minimization of synchronization barriers: the first two topics because such parallelism can not be easily and

[2] URL `http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/SOFT/TransTOOL/`

efficiently implemented in a data-parallel language like HPF (it is easier to exploit it when *compiling* HPF), the third topic because HPF codes are usually compiled into SPMD codes, synchronized by nature.

Our main goal is to expose to the programmer the maximal parallelism that can be detected. We are interested only in understanding if a large set of independent computations can be detected, and if they can be described by parallel loops. In other words, we aim at detecting loops that, in HPF, can be preceded by the directive `!HPF$ independent` (denoted by DOPAR in the pseudo-code below).

Fine-grain Parallelism

In many applications, there is no need to use sophisticated dependence analysis techniques and parallelization algorithms for detecting full parallelism. A simple algorithm such as Allen and Kennedy's algorithm [3] is sufficient for most of the loops. Our implementation of Allen and Kennedy's algorithm will be used as a comparison base to evaluate how often more sophisticated algorithms are needed. In this context, we recently showed that, as long as dependence level is the only information available, Allen and Kennedy's algorithm detects maximal parallelism (see [23]).

In some loops however, some more accurate representation of dependences is needed. Techniques based on the hyperplane method [43] have been developed in the past so as to exploit a more accurate description of dependences such as the description by direction vectors (see Wolf and Lam's algorithm [59]). This last algorithm is able to take into account the information given on all components of distance vectors (which is not possible with Allen and Kennedy's algorithm and level of dependences), but it is not able to use the information concerning the structure of the dependence graph (which is the basis of Allen and Kennedy's algorithm for applying loop distribution).

We thus proposed a novel algorithm, Darte and Vivien's algorithm, that combines and subsumes both algorithms [24]. We found that this algorithm exploits optimally the structure of the graph and the information on direction vectors. It is even optimal for a more accurate representation of dependences that we called PRDG (polyhedral reduced dependence graph), roughly speaking, approximations of dependences by non parameterized polyhedra, defined by vertices, rays and lines.

Medium-grain Parallelism

In HPF, codes with single innermost parallel loops are often not parallel enough to offer good performance. In this case, the grain of parallelism must be increased, either by trying to move up the parallel loop to the outermost possible level, or by using blocking (tiling) techniques.

We studied this tiling problem in [13] in the simple case of uniform loop nests, and it turns out that Darte and Vivien's algorithm can be easily adapted to the tiling technique, as Wolf and Lam's algorithm that was developed with a "tiling spirit". Actually, the detection of parallel loops and the detection of maximal tiling, related to maximal sets of permutable loops, are two equivalent problems.

We are currently implementing in TransTOOL, a tiling version of Darte and Vivien's algorithm: it informs the programmer of the maximal parallelism he can hope, and proposes loop transformations that reveal maximal parallelism, either as fine-grain parallelism, or as medium-grain parallelism.

A lot of problems remain to be solved such as the choice of the block size for tiling, the choice of a suitable mapping (with possibly temporary arrays), ... As the reader can notice, the above algorithms are able to generate `!HPF$ independent` directives, but they do not address the generation of directives such as align or distribute. This is still left to the programmer: he has to choose the mapping that exploits at best the parallelism that has been detected.

We conclude this section by a very simple example, that illustrates the type of codes that can be generated by our parallelism detection algorithm. Figure 1.3(a) shows the original code, and Figure 1.3(b) the code with fine-grain parallelism. Note that Allen and Kennedy's algorithm would also find one parallel loop in this example. However, the fact that loop distribution can be avoided cannot be found by Allen and Kennedy's algorithm.

```
DO i = 1, n
  DO j = 1, n
    a(i, j) = a(i - 1, j + 1) + b(i - 1, n)
    b(i, j) = a(i, j - 1) + b(i - 1, j - 1)
  ENDDO
ENDDO
```

(a)

```
DO i = 1, n
  b(i, 1) = a(i, 0) + b(i - 1, 0)
  DOPAR j = 2, n
    a(i, j - 1) = a(i - 1, j) + b(i - 1, n)
    b(i, j) = a(i, j - 1) + b(i - 1, j - 1)
  ENDDOPAR
  a(i, n) = a(i - 1, n + 1) + b(i - 1, n)
ENDDO
```

(b)

Figure 1.3: Original code and code with fine-grain parallelism.

1.4.2 Detection of Pipelined Loops and Code Generation

Parallel distributed memory machines improve performance and memory capacity but their use adds an overhead due to the communications. To obtain programs that perform and scale well, this overhead must be minimized. Part of the job is devoted to communication libraries, which should provide efficient point-to-point and macro-communications. Another important issue is to "hide" communication as much as possible, by overlapping them with independent communications.

Asynchronous communications can be used to overlap computations and communications. The call to the communication routine (send or receive) is then issued as soon as possible in the code. A wait routine is used to check for the completion of the communication. Unfortunately, this is not always legal due to the dependences between computations and communications. Pipeline schemes are also sometimes found within the code. These schemes lead to a sequentiality in the execution of the whole algorithm.

The optimization we have added is what we call *Macro-pipeline Overlap*. There is a sequentiality within the code (see Figure 1.4 (A)). Processor P1 must wait for processor P0 to complete his computation and send the results, to receive the data and start to work. As soon as it has finished, it sends the results to processor P2 which, in turn, starts to work on the received data. The total execution time is higher than the sequential one because of the overhead of the communications. One first solution is to start the communications as soon as possible, i.e. as soon as one processor has computed one data item. For each data item computed, an other one is sent to the following processors so they can start as soon as possible. This is called a fine-grain pipeline ((B) on Figure 1.4). This solution adds an overhead because of the communication startup time. This time is usually higher than the cost of the communication of one element. Thus, the total time can be higher than the one without pipelining. A trade-off has to be found which minimizes the execution time. This is a coarse grain pipeline ((C) on Figure 1.4).

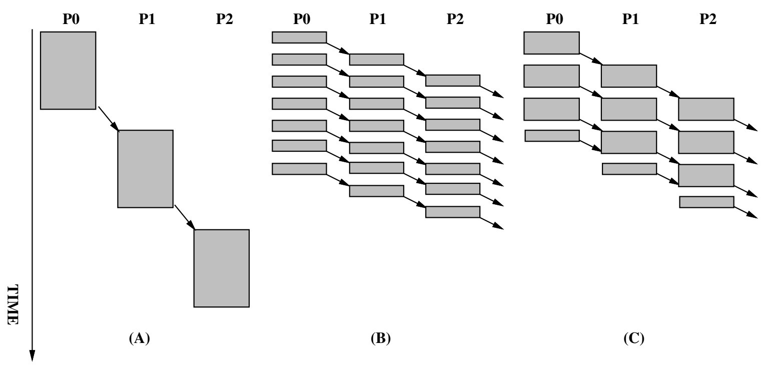

Figure 1.4: Macro-pipeline.

A typical example of a code that may benefit from a macro-pipeline optimization is the ADI algorithm given on Figure 1.5.

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A, B
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
      ...
!     sweep along the columns
      DO I = 2, N
        DO J = 1, N
          A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
        END DO
      END DO
!     sweep along the rows
      DO J = 2, N
        DO I = 1, N
          A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
        END DO
      END DO
```

Figure 1.5: High Performance Fortran Version of the ADI algorithm.

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A, B
!HPF$ DISTRIBUTE (*,BLOCK) :: A, B
      ...
      DO I = 2, N     ! parallel execution
         DO J = 1, N
            A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
         END DO
      END DO

      CALL DALIB_LOCCS_DRIVER (BLOCK, 2, 0,
           A(:,2:N), [0,1], B(:,2:N), [0,0])
      ...

      EXTRINSIC (HPF_LOCAL) SUBROUTINE BLOCK (A, B)
      REAL A(:,:), B(:,:)
!HPF$ DISTRIBUTE *(*,BLOCK) :: A, B
      DO J=lbound(A,2),ubound(A,2)
        DO I=lbound(A,1),ubound(A,1)
           A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
        END DO
      END DO
      END
```

Figure 1.6: ADI algorithm using the LOCCS library.

We have designed a library for the optimization of pipelined computations called the LOCCS [26, 28] [3]. This library has been integrated in the ADAPTOR compiler [16] and we are currently working on its integration within TransTOOL.

Within ADAPTOR, the LOCCS library consists in a driver routine that takes as parameters information about the distributed matrices, the distributed dimension(s) and a routine which is called at each step of the macro-pipeline. Within the driver routine, choices are made to use macro-pipelining or not, and also on the way of doing this optimization. There are several reasons to use a library instead of generating the code directly in the SPMD source code. The first one is the ease of use for the programmer of an HPF compiler. Instead of generating several lines of code, the compiler only has to generate a subroutine call, to fill the parameters and to generate the computation routine. Another reason is to be able to perform run-time optimizations like, for example, the dynamic computation of the optimal grain size as a function of the network load, cache effects, and so on.

We have obtained very good results using this library in the ADAPTOR compiler, for example with the ADI algorithm given in Figure 1.5. There are two strategies to solve this problem, one using a redistribution (transposition) and the other one using our library to have an optimized pipelined execution. The pipelined execution achieves nearly the optimal speed-up and a dynamic data remapping is not necessary in this case. Other applications of the library will be given in [17].

Now we need to integrate the LOCCS inside TransTOOL. First we need to find what Tseng called *Cross Processor Loops* (CP loop) in [56]. A loop is a CP loop if it has a true dependence carried by the loop and of course if its iteration crosses the processors boundaries. If a loop is a CP loop, one processor needs the results of the computation of its left or right neighbor to start to work. However, telling that a loop is a CP loop is not sufficient to say that a macro-pipeline execution is efficient. We are working on an algorithm that detects loops that can benefit from a macro-pipeline execution.

For the computation of the optimal granularity of the pipeline, we will use the OPIUM library [27]. The granularity will be also tuned at run-time depending of cache effects or network trafic (this has also been suggested in [1] and [53]).

The user will be able to give to TransTOOL the parameters of the target machine (parameters of a communication, costs of an average computation). These parameters will be used for the optimization of the code generation. For example, when using PVM on a network of workstations connected via Ethernet, no macro-pipeline should be used because of the huge costs of communication startups.

[3] Low Overhead Communication and Computation Subroutines.

Chapter 2

Data-Structure Visualization and HPF Extensions for Irregular Problems

2.1 Data Visualization and Trace Analysis

When using a data-parallel language, the user mainly conceives his application in terms of data. For instance, when talking about his program, he would say "... *block 1 of matrix A is added to block 1 of matrix B* ...". Hence, when considering the behavior of an application, there is hardly any reason for displaying process-based information, even though the effective implementation, i.e. the result of the compilation, is eventually executed using the model of communicating processes. Therefore HPFIT will offer a set of software components, VisIt, developed at the LaBRI, that will make it possible to visualize and analyze data-parallel programs and their behavior in terms of the data they work on. As of writing, some of these software components are available as prototypes. This chapter describes one of them, called DDD, and its underlying model.

2.1.1 State of the Art

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[8, 9, 11]). Both hardware and software vendors now supply environments of their own. Furthermore, public domain tools (such as ParaGraph[33]) are now being delivered either as fully supported or public domain ported tools. In terms of performance measurement, these tools that were used when dealing with message passing applications can still be used. Nevertheless, there is a lack of relationship between the display they provide and the semantics of the application. If the message passing model was close to the execution model on distributed memory machines it is no longer the case when considering data parallelism.

Although research is now on its way at various places, there is still a lot to be done in terms of tools that would help users to tackle the paradigm of data-parallel programming. One of the reasons why this is so, is that tools need information that cannot always be accessed easily. Consider for instance, information regarding distributions of arrays. 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[32]. 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 project called *Distributed Array Query and Visualization* (DAQV [46]). The aim of this project is to provide an environment to query and visualize distributed arrays. Among the participants of this projects are the researchers formerly involved in the DDV project[30]: Data Distribution Visualization (for performance evaluation). The main goals of this project are: to get information about data decompositions using a dedicated language called ADL (Array Definition Language); to query the arrays using a language called AQL (Array Query Language) usable both by means of a graphical interface and an API; to visualize arrays using tools such as IRIS Explorer. This implies to setup definitions: definition of services; definition of how data can be accessed; definition of a protocol to dialogue with/between tools. A prototype implementation, based on PC++[12] with visualization achieved by means of IRIS Explorer, has been around for some time. An HPF implementation has recently been released.

An other very interesting project is EPPP[36]. It offers a development platform, ranging from the language to a performance debugger. The language underlying the environment is HPC which is a dialect of C augmented to meet the data parallel paradigm. There are many tools which are offered in EPPP for which we plan to have similar systems in HPFIT. The main difference is that we have definitely chosen HPF as our expression language. This makes a big difference because it implies that we will not really control the run-time support as it is the case for EPPP, which controls the run-time libraries of HPC. For instance, this will make it really more difficult for us to get run-time information regarding data distribution or exchanges, still being independent of the HPF compiler at hand.

Our approach aims, as far as possible, at being independent both from the user and from instrumentation of runtime libraries. We only rely on source preprocessing, which, provided the language obeys a standard, does not depend on anything.

There are many other tools or systems which we have studied. We will not describe all of them in this paper because they seem less related to our goal (see for instance EPPP[36], P2D2[21], Pharos[55], Paradyn[47]...).

One of the other aims of HPFIT, hence of VisIt, at least in its second release is to tackle sparse and irregular data-structures. We have not found any tool that would deal with this paradigm.

2.1.2 A Model and its Validation on Data Distribution

In the current prototype, we first have concentrated, for two main reasons, on data distribution.

- When willing to tune HPF codes, the distribution of data is an important factor. This parameter has to be correctly evaluated to obtain interesting performances at the end of the compilation chain. Using the possibilities of alignments and distributions, the final code becomes very complex and it is really hard to have an idea of how matrices are distributed among a virtual network of processors.

- The problem of visualizing and querying data distributions is simple enough, so that we can still concentrate on the underlying model which we are setting up, still providing concrete results and utilization of the model (which is a way to make a first evaluation of it).

Abstraction from Implementation

When designing our model, we took into account two criteria which we consider central within VisIt, i.e. within HPFIT:

- Although most data-parallel applications use some sort of matrices, these matrices might code other data-structures, trees for instance. In such a case, there is no point in providing the user with a representation of the matrix ; he better wants to see a tree.
  This is an attempt to take the semantics of the application into account.

- A tool, to be general purpose, cannot rely on any assumption about the implementation of a data-structure: for instance, a matrix can be coded by lines, columns, blocs, and such. Furthermore, even though the matrix is implemented by lines, the user should be presented with a matrix as a whole, not with a set of lines.
  This is an attempt to abstract from implementation.

Therefore, we have set up a model to represent data structures that makes it possible to abstract from both their effective implementation, and their meaning. In this section, we give a brief overview of this model.

The basic idea is that any data structure can be represented by a graph, the nodes of which are the items contained inside the data structure. Using this model, a data structure is fully described by means of:

a root: it is the root of the graph, i.e. the entry point of the data structure,

a degree d: it is the maximum number of successors of any node of the graph,

d successor functions s_i: these functions give, for any node, its successors in all directions. The i_{th} successor of node N can be noted as $[0, 0, ..., 1, ..., 0, 0]$ where the only 1 is at position i.

For instance, a dense matrix $matrix[2][3]$ would be described very easily, since its successor functions effectively match index incrementations (see Figure 2.1).

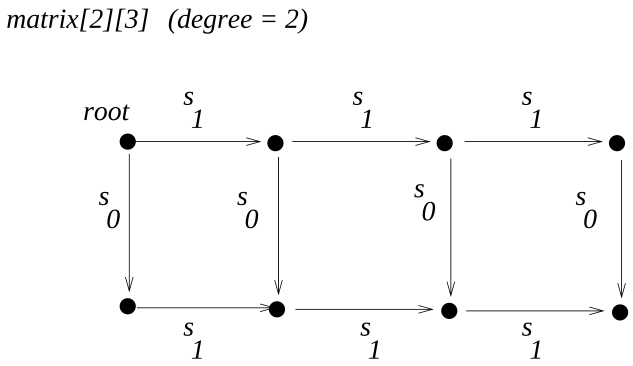

Figure 2.1: A graph coding a matrix.

Assuming this graph is provided, either from a library, or by the user, there is no need to know how the data structure is effectively implemented. This dense matrix could be coded by rows, columns, and such. The successor functions build a mapping from an implementation to a logical model of the data-structure. Furthermore, if a tree were coded in the matrix, the successor functions could be used to describe it, providing a tree model of a tree implemented by means of a matrix.

This model, and the notation that we have introduced to express successors lead to an array-like way to access items in the data structure. For instance, $matrix[1, 2]$ would be represented by $root[1, 0][0, 1][0, 1]$. Note that in this case, this notation is not unique, e.g. we could for instance have used $root[0, 1][1, 0][0, 1]$. This comes from the fact that, in this specific case, we have an equivalence between $N[1, 0][0, 1]$ and $N[0, 1][1, 0]$, where N is a node. A model having this property will be referred to as *commutative*. In such a case, we can use the compact notation $[1, 2]$ built from flattening $A[1, 0][0, 1][0, 1]$. We then get the same notation as we have when dealing with standard matrices.

DDD: Data Distribution Visualization

The software component of HPFIT that implements this model to visualize and query distributions is called DDD or D^3 for ***D****ata* ***D****istribution* ***D****isplay*, and is part of VisIt. When writing this paper, it is still a prototype, but nevertheless DDDmakes it possible to visualize the distribution of data over templates and virtual processors. This is achieved using *mapping functions* built on top of the model described above, which, being given an index in a source object, provides the corresponding index in the target object:

```
Index map(Index src);
```

Within the current prototype, we have used a one-to-one mapping, but of course a one-to-n mapping will be implemented. It simply consists in having a mapping function that returns either a list or an enumeration. Furthermore, since we intend to answer user queries such as "which items are mapped to this processor?", we will implement what we call *reverse mapping functions* (see Figure 2.2). These functions will be used to move back from mapping targets to mapping sources.

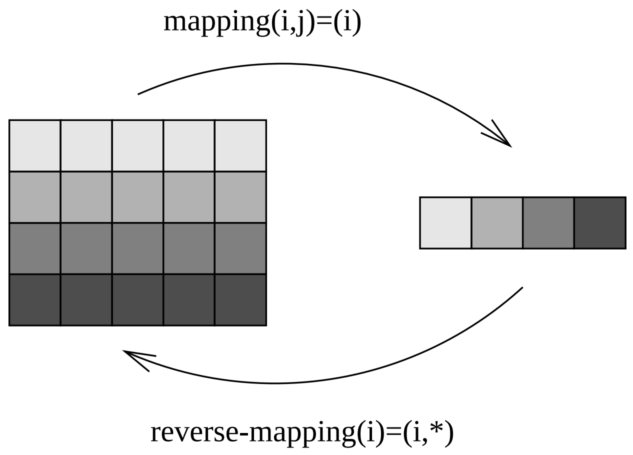

Figure 2.2: Mapping and reverse-mapping functions.

Once again, these functions will, as far as possible, be extracted from the application at preprocessing or at compile time (using the `DISTRIBUTE`, `ALIGN` directives, and such), which is possible when considering current HPF distributions.

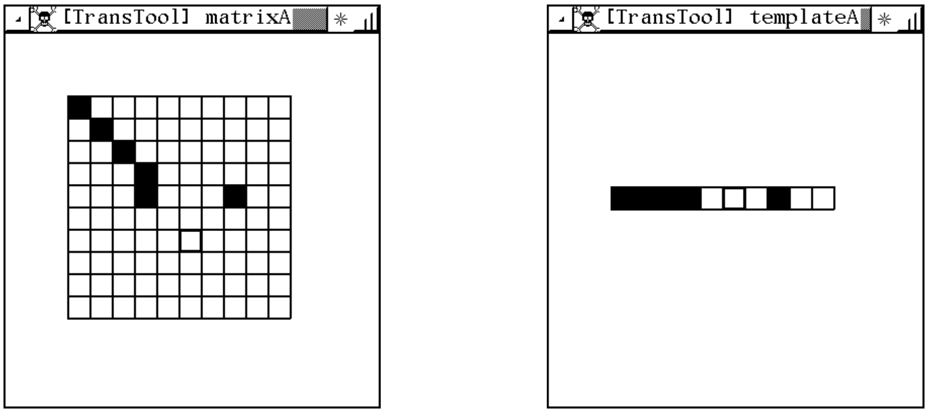

Figure 2.3: A snapshot of **DDD**.

Figure 2.3 shows a snapshot of the current DDD prototype implementation, which although not yet user-friendly makes it possible to validate the model.

The reason why the model might look a bit complicated for standard dense matrices, is that it is designed so as to provide support for sparse data structures. Of course, dealing with sparse and irregular data-structure also relies on language level support. This last point is the main topic of the next section.

2.2 HPF Extensions for Irregular Problems

As presented here, HPFIT does not solve every problem encountered during the parallelization of applications. Many scientific applications use sparse matrices or other hierarchical and dynamic *irregular* data structures that must be represented by usual arrays that hide the functionality of the code and prohibit its optimization. Currently, it is impossible to write such codes in HPF that can be compiled efficiently for distributed memory machines. An other study, which uses HPFIT as a development plateform, is based on a new data structure "Tree". This data structure, which is intended for the representation of a wide class of irregular data structures, allows much more convenient compiler optimizations that might result in efficient execution also on MIMD machines. This last study which is presented in this section is a joined project between the LaBRI, the GMD/SCAI and the LIP.

Large sparse systems of linear equations and other hierarchical and dynamic data structures occur in many scientific and engineering applications encountered in military and civilian domains (fluid dynamics, structural mechanics, ...) [58]. They arise for example when performing a finite element method on unstructured 2D or 3D meshes.

These irregular data structures must be represented by some arrays (e.g. see SPARSKIT [52]). Unfortunately, the use of these arrays for the compact representation of sparse matrices make the code difficult to read, hide the functionality of the code (e.g. due to the indirect addressing) and prohibit the optimization of the code. Currently, it is impossible to write such codes in HPF that can be compiled efficiently for distributed memory machines.

Vienna Fortran [57] proposes simple language features that permit the user to characterize a sparse matrix and to specify the associated representation. The compiler utilizes this information to identify the use of the sparse matrix and to apply optimizations. The advantage of this approach is that the user has only to add some directives or declarations in his code and he does not loose the portability. But on the other side, the complexity of the compiler increases dramatically. The compiler must know all the possible representations (e.g. all the different representations of SPARSKIT) and also identify every use of the sparse matrice within the users code.

Bik and Wijshoff [10] have implemented a restructuring compiler which automatically converts programs operating on dense matrices into sparse code. This approach is very convenient for the user, but in most situations the compiler fails to find a good and efficient representation.

Our approach is based on a new data structure that is called "Tree". This data structure is mainly intended for the representation of sparse matrices, but can also be used for any other hierarchical data structure. The advantage of the new data structure for the user is the better readability of his programs though he has to rewrite existing code. But the data structure itself will allow much more convenient compiler optimizations that might result in efficient execution on MIMD machines. The HPF directives provided for the mapping of arrays (distribution and alignment) and access of array elements can be used in the same way for trees. The compiler can use regular data dependence checking and perform standard optimizations.

This chapter is organized as follows. In Section 2.2.1 we present the idea of the new data structure "Tree" and how a tree can be used for the representation of sparse matrices and other hierarchical data structures. It follows the description of how trees can be embedded and implemented in FORTRAN. Section 2.2.5 shows the use and benefits of the trees for a HPF programming of a sparse Cholesky factorization algorithm.

2.2.1 Trees

We first present the idea of how trees can be used for representing sparse data structures and other hierarchical informations.

Description of Trees

A tree is a connected, directed and acyclic graph. The *root* of the tree is the node that has no predecessors. The *leaves* of the tree are the nodes that have no successors. A node is on the *level* p ($p \geq 0$) if its distance to the root is p. The *height* h of the tree is given by the maximal level number that exists. Figure 2.4 shows a tree of height 2.

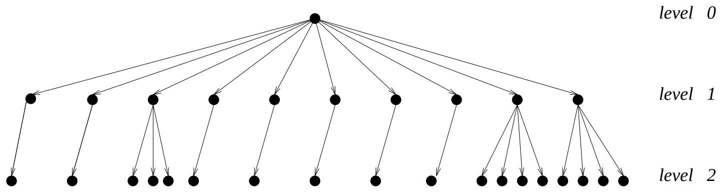

Figure 2.4: Tree of height 2.

A tree can be used to represent information that is hierarchically organized. The access to a node in the tree is specified by the path from the root (level 0) in the tree. Compared with arrays, a tree is the more compact representation of information with varying number of entries.

Representation of Sparse Matrices with Trees

For a sparse matrix A, sparsity can be exploited to save *storage requirements* by only storing the nonzero elements. Storage required to store the numerical values is called *primary storage*, while storage that is necessary to reconstruct the underlying matrix is referred to as *overhead* storage.

Many different ways of storing sparse matrices have been devised to take advantage of the structure of the matrices or the specificity of the problem from which they arise. In a similar way, there are many possibilities of representing sparse matrices with trees. We give an example below.

Example 2.2.1 *CSC representation of sparse matrices with trees.*

The Compressed Sparse Column (CSC) format is one of the many possibilities of representing sparse matrices (SPARSKIT [52]). The nonzero elements are traversed by the columns of the matrix where for each element the number of the row will be kept in an additional array. For this scheme, the sparse matrix can be represented by a tree with three levels.

- the i-th node on level 1 represents the i-th column.
- the j-th son of the i-th node on level 1 represents the j-th non-zero element of column i. Each such node is a leaf and has an entry for the value of the matrix and an entry for the number of the row to which the element belongs.

Figure 2.4 represents the tree associated with the sparse matrix given at Figure 2.5.

	1	2	3	4	5	6	7	8	9	10
1			19						54	69
2	53							12		
3									44	37
4					17					64
5			34			93			19	
6				72						27
7		21					13			
8			16						23	

Figure 2.5: A sparse matrix with 10 columns, 8 rows and 18 non zero coefficients.

Another Example

Example 2.2.2 *Representation of a finite element mesh.*

A finite element mesh [54] is a set of compatible elements, each of them composed of some nodes given in a predefinite order and associated with the unknowns that must be computed. This computation works in two steps. In the first step, one computes the finite element matrix and the right-hand side (*assembly algorithm*), and in the second one, we solve the sparse system of linear equations. The assembly step consists in a global loop over the set of elements; for each one, one performs a loop over the nodes of the element in order to accumulate its contributions to the finite element matrix. In parallel, the loop over the elements is distributed, so the data structure describing the mesh must provide an *element-node* relation. Figure 2.6 shows how trees can be used to deal with this relation.

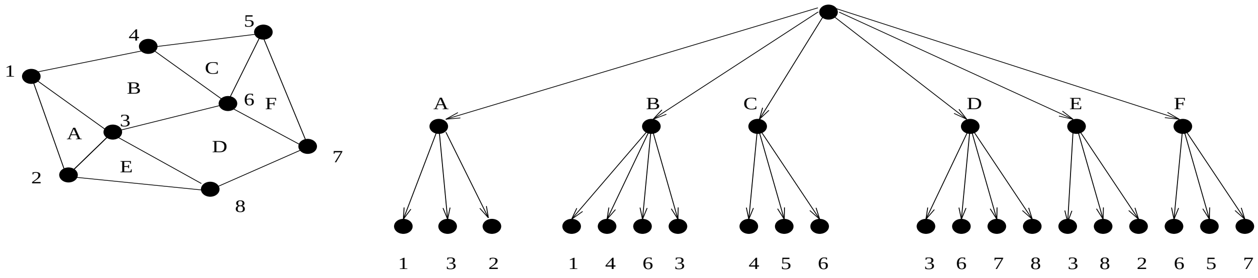

Figure 2.6: Tree representation of element-node relation mesh.

2.2.2 Using Trees in Fortran

This section describes how the data structure "Tree" can be embedded in the programming language `FORTRAN`.

Definition of Trees in Fortran

Within Fortran, a tree will be defined by specifying a datatype for every level of the tree (except level 0). We will use the derived types already provided in Fortran 90 to specify these datatypes. Each node of the tree at a certain level will have an incarnation of the datatype specified for this level. The following declaration statement defines a tree named `A` of height h where every $type_i$, $i > 0$, is the name of a derived type.

`TREE` ($type_1$, $type_2$, ..., $type_h$) `A`

Trees are allocated and deallocated like allocatable arrays in Fortran 90 by the `ALLOCATE` and `DEALLOCATE` statement. For the allocation of a tree, each node of each level must be specified how many sons it has.

```
ALLOCATE (A(A_SIZE1, A_SIZE2(:), A_SIZE3(:), ...), ...)
```

The number of sons of the root, i.e. the number of nodes on level 1, is given by `A_SIZE1` which is a scalar integer value. The following arrays specify always the numbers of sons for each node on the next level.

After the allocation the number of nodes cannot be changed.

Example 2.2.3 *Definition of a tree for representing a sparse matrix with a CSC format.*

```
TYPE LEVEL1          ! no data on level 1
END TYPE

TYPE LEVEL2
   INTEGER ROW       ! line number
   REAL VAL          ! non zero value
END TYPE
```

```
TREE (LEVEL1, LEVEL2) A

ALLOCATE (A(10, [1, 1, 3, 1, 1, 1, 1, 1, 4, 4]))
```

This allocation will create a tree corresponding to Example 2.2.1. Each node on level 1 has its own values for `ROW` and `VAL`.

Access of Tree Elements

The following notation is used to specify a certain node on level k, $1 \leq k \leq h$, in the tree:

```
A(i1,i2,...,ik)
```

The numbering of sons starts always with 1. Every index specifies the *relative position* of the node within the childs of the node one level above. The array notation of Fortran 90 can be used to specify a list of nodes where the number of indexes exactly specifies from which level the nodes will come.

```
A(3)         ! node 3 on level 1
A(9,:)       ! all sons at level 2 of node 9 on level 1
A(5:7)       ! nodes 5 to 7 on level 1
A(5:7,:)     ! all sons at level 2 of nodes 5 to 7 on level 1
```

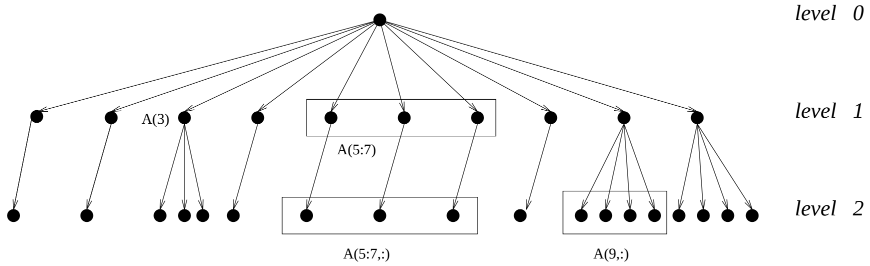

Figure 2.7: Accessing tree elements

The elements of a derived type for a node in the tree can be accessed with the `%` notation of Fortran 90.

```
A(i1,i2,...,ik)%component
```

This variable denotes the incarnation of the variable `component` at the corresponding node. `component` must be a component of the derived type specified for level k.

2.2.3 Intrinsics for Trees

Two intrinsic functions that give some information on a tree are defined. The intrinsic function `height` returns for a node the height of the subtree rooted at this node. So for the tree of Figure 2.7, `height(A)` returns the value 2, and `height(A(3))` returns the value 1. The intrinsic function `size` delivers the number of sons of a certain node. This routine is very helpful to write loops for traversing the trees.

```
size (A(i1,...,ik))
```

If a second scalar integer argument `p` is provided, the function delivers the number of sons `p` levels deeper. So, `size (A(i1,...,ik),1)` is equivalent to `size (A(i1,...,ik))`.

Example 2.2.4 *Initialization of a sparse matrix described by the tree defined in Example 2.2.3.*

```
DO column = 1, size(A)
   DO row = 1, size(A(column))
      A(column,row)%ROW = ...     ! set the row position
      A(column,row)%VAL = ...     ! set the nonzero value
   END DO
END DO
```

Mapping of Trees

The mapping of a tree is defined by *the mapping of a certain level of the tree*. The level of a tree can be specified by using the array notation, like `A(:)` for the first level, `A(:,:)` for the second one. The level of a tree is considered as a one-dimensional array and its mapping is specified via the HPF directives.

```
!HPF$ DISTRIBUTE A(:) (BLOCK)
!HPF$ DISTRIBUTE A(:,:) (CYCLIC)
!HPF$ ALIGN A(:) (I) WITH X(I)
```

If a tree is distributed along the first level, the nodes on level 1 will be distributed among the available processors (see Figure 2.8 with 3 processors). If a tree is distributed along the second level, all nodes on level 2 will be distributed among the available processors (Figure 2.9).

The distribution of a level implies a replication of all levels from the root until to this level. The higher levels are distributed according the distribution of the given level. If a tree is distributed along a certain level, it is guaranteed that all sons of a node on this level will reside on the same processor (collapsed).

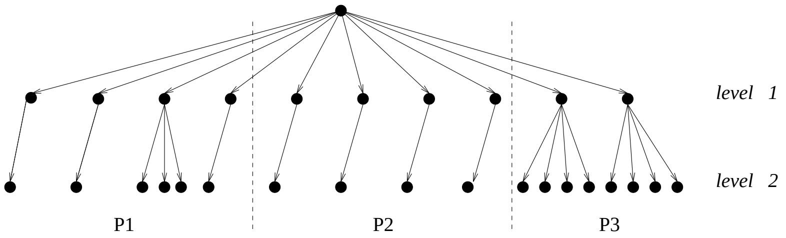

Figure 2.8: Block distribution of a tree along the first level.

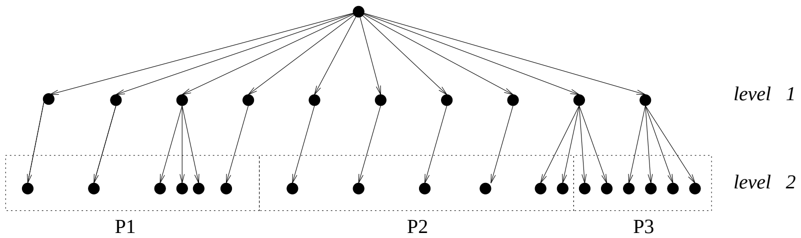

Figure 2.9: Block distribution of a tree along the second level.

The HPF syntax is intended in the following way:

```
H307 distributee  :: tree-level

H316 alignee      :: tree-level
```

```
H321 align-target :: tree-level

N001 tree-level   :: tree_name (level-ind-list)
N002 level-ind    :: ':'
```

As many algorithms could benefit of an irregular distribution of the tree, it is helpful if the HPF compiler supports irregular distributions [34]. If this feature is available for arrays, it should be possible to use it also for trees.

2.2.4 Implementation of Trees

In the final code, trees will be represented by arrays. We propose the following solution that is very similar to the array representations of SPARSKIT:

- Every level of the tree specified by a data type will be represented by one or several corresponding arrays that contain the incarnations of the derived type.
- For every level i, $0 \leq i < h$, h is the height of the tree, exists an array containing the number of sons for every node; if $j = i + 1$, this array will be denoted `tree-name_SIZEj`. For $i = 0$, `tree-name_SIZE1` is a scalar integer value.
- For every level i, $1 \leq i < h$, exists an integer array that contains an offset for every node of this level. This offset gives for the node the position of his first son within the array for the next higher level; each such array will be denoted `tree-name_PTRi`.

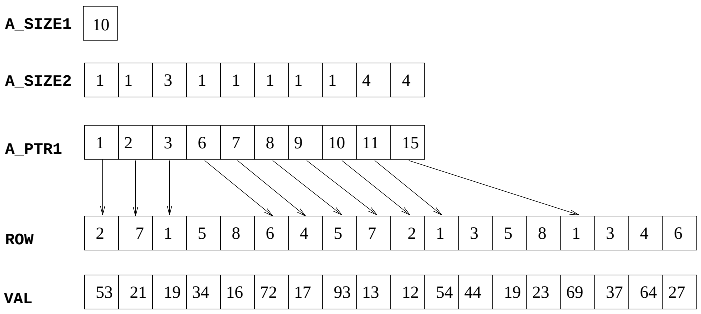

Figure 2.10: Set of arrays used for the representation of the tree named A of Example 2.2.1.

The distribution of a level (see previous section) results in a corresponding distribution of the arrays for this level and the arrays of the higher levels. A block or a general-block distribution of a level implies a general-block distribution of the higher levels. A cyclic or block-cyclic distribution of a level might imply irregular distributions of the higher levels.

2.2.5 Examples

The Cholesky factorization of sparse symmetric positive definite matrices is an extremely important computation arising in many scientific and engineering applications. However, this factorization step is quite time-consuming and is frequently the computational bottleneck in these applications. Consequently, it is a significant interesting example for our study. The goal of the sparse Cholesky computation is to factor a symmetric positive definite $n \times n$ matrix A into the form $A = LL^T$, with L lower triangular.

Two steps are typically performed for the computation. First, we perform a *symbolic factorization* to compute the non-zero structure of L from the *ordering* of the unknowns in A; this ordering, for example using *nested dissection strategy* must reduce the *fill in* and increase the parallelism in the computations. This (irregular) data structure is allocated and its initial non-zero coefficients are those of A. In the pseudo-code given below, this step is implicitly contained in the instruction `L = A`. Second, the *numerical factorization* computes the non-zero coefficients of L in the data structure. We refer to [29] for more details.

We focus here on the second step which is the most time-consuming. We describe in the following the implementation of one formulation of Cholesky factorization using our new data structure and new notations.

Sparse Column Algorithm

The chosen formulation is a column-oriented one (see for example [6, 50] and included references).

1. `L = A`
2. `for k = 1 to n do`
3.   `for i = k to n with` $l_{ik} \neq 0$ `do`
4. $l_{ik} = l_{ik} / \sqrt{l_{kk}}$
5.   `for j = k+1 to n with` $l_{jk} \neq 0$ `do`
6.   `for i = j to n with` $l_{ik} \neq 0$ `do`
7. $l_{ij} = l_{ij} - l_{ik} * l_{jk}$

The Algorithm with Trees

The symmetric sparse matrix is represented by the CSC format described by the tree defined in Example 2.2.3. In the following of this example, we suppose that the first level of the tree, and in this way the columns of the matrix, are distributed in a cyclic way.

Figure 2.11 shows how coefficients in column `J` (with `J = L(K,alpha)%ROW`) are modified by coefficients in column `K`.

```
!HPF$ DISTRIBUTE L(:) (CYCLIC)

      DO K = 1, size(L)
         L(K, 1:)%VAL = L(K, 1:)%VAL / SQRT (L(K,1)%VAL)
         DO alpha = 2, size(L(K))
            J = L(K,alpha)%ROW
            beta = 1
            DO gamma = alpha , size(L(K))
!              find beta with L(J,beta)%ROW = L(K,gamma)%ROW
               I = L(K,gamma)%ROW
               DO WHILE (L(J, beta)%ROW .ne. I)
                  beta = beta +1
               END DO
               L(J, beta)%VAL = L(J, beta)%VAL - L(K, alpha)%VAL *
                                L(K,gamma)%VAL
            END DO   ! loop gamma
         END DO    ! loop alpha
      END DO     ! loop K
```

HPF Parallelization

The iterations of the loop with the index `alpha` are independent (loop corresponding to the instruction `5` in the pseudo-code). One iteration should be executed by the owner of the column `J=L(K,alpha)%ROW` to minimize the communication. One could specify the ownership of the iterations by using the `HOME` clause for tree nodes that is proposed for HPF 2.0 [34].

```
!HPF$ INDEPENDENT, NEW(I, J, beta, gamma), ON HOME L(L(K,alpha)%ROW)
      DO alpha = 2, size(L(K))
         ...
      END DO
```

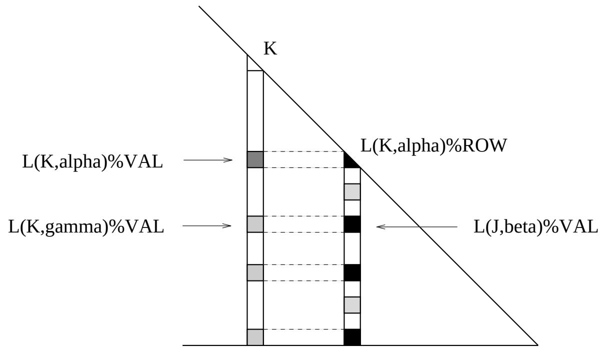

Figure 2.11: Modification scheme and dependences between columns.

We assume that the compiler will replicate the entry `ROW` of the second level before the outermost loop to avoid communication. As the second level is implicitly distributed by the cyclic distribution of the first level, the compiler generates a temporary array that contains the replicated data. Within the parallel loop, only the terms `L(K,alpha)%VAL` and `L(K,gamma)%VAL` can cause communication. By the notation of tree indexes, the compiler will be able to identify these two terms. The compiler must extract the communication before the parallel loop which might not be possible as the compiler has not sufficient information about the dependences within the loop. We propose the clause `COMM IN` to advise the compiler, that the owner of column `K` can send the data `L(K,:)%VAL` to the owner of the corresponding iterations.

```
!HPF$ INDEPENDENT, ON HOME L(L(K,alpha)%ROW), COMM IN (L(K,:)%VAL)
      DO alpha = 2, size(L(K))
         ...
      END DO
```

2.2.6 Irregular Distributions

To illustrate the use of our data structure and notations, we have utilized a cyclic distribution for the trees.

```
!HPF$ DISTRIBUTE L(:) (CYCLIC)
```

In fact, for an efficient parallel execution of the solvers, one must use a specific distribution that is an irregular distribution [50, 51]; this "optimal" distribution can be given by an integer array `OWNER` and can be computed by a preprocessing algorithm or at runtime.

```
!HPF$ DISTRIBUTE L(:) (INDIRECT(OWNER))
```

Conclusion and Future Work

In this report, we have presented the first versions of the HPFIT and TransTOOL projects. HPFIT will provide one interface to many other tools used in the parallelization of applications like performance monitors, trace analyzers, and simulation tools. In this first version, HPFIT is not a totally new environment built from scratch but an "intelligent" interface into which existing or new tools are being plugged. Thanks to this interfacing, we will be able to add new tools as they appear.

Data distribution and alignment is one of the most important problem for the parallelization of applications using HPF. It is known to be a NP-complete problem in most cases; however for classical problems, heuristics can be found that will lead to good performance. Thus we need to add a tool for semi-automatic data distribution (within HPFize). Another problem that has already been raised by Kennedy et al. in [35] is the interaction between the HPF compiler and the editor. This is not a trivial work as the compiler can make huge transformations to obtain an SPMD code with local arrays and calls to communications routines.

Converting dusty F77 to Fortran 90 seems to be a useful intermediate step in the parallelization process. For example, data parallelism could be expressed by array syntax and FORALL loops (Fortran 95). Part of this work is clearly not our job (cleaning F77) but we could add some fonctionnalities to integrate F90 constructs in the HPFize part of TransTOOL, by taking those loops nests that can be transformed.

In the second part, we have presented a data structure visualization tool and an HPF support for irregular data structures.

Future research directions concerning the visualization tool are the followings. The ability to plug in visualization modules will be based on the interconnection of basic, possibly distributed, modules. Some systems are already based on such a mechanism, but usually execute in a centralized framework. See for instance the Pablo programming environment[49] or the DAQV project[31]. The development of more mechanisms to support sparse matrices and other data-structures.

In Section 2.2, we have presented the new data structure "Tree" and show how it can be used for the representation of sparse matrices. Scientific codes working on sparse matrices represented by trees offer the possibility to be compiled efficiently by a HPF compiler that supports this new data structure. We will extend the HPF compilation system ADAPTOR [19] to support the tree data structure. With the current compilation technology, it might be difficult to identify always the most efficient communication in parallel loops and statements. This is especially true for the use of trees. We would like to introduce *directives to aid the compiler in generating the most efficient communication.* HPF 2.0 provides support for indirect distributions of arrays. This will also be very useful for the distribution of trees. We will investigate the advantages for certain applications.

A lot of work remains to be done in the field of tools for semi-automatic parallelization of applications. We hope that a collaboration between several laboratories will lead to an interesting and performant tool made available to the whole community.

Acknowledgments

We would like to thanks Gilles Lebourgeois, Olivier Reymann, Georges-André Silber, Lionel Tricon and Julien Zory for their work within the TransTOOL project.

Bibliography

[1] V.S. Adve, C. Koelbel, and J. Mellor-Crummey. Compiler Support for Analysis and Tuning Data Parallel Programs. Technical Report CRPC-TR95520, Center for Research on Parallel Computation, Rice University, March 1995.

[2] V.S. Adve, J.C. Wang, J. Mellor-Crummey, D.A. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environnement for Data Parallel Programs. Technical Report CRPC-TR94513-S, Center for Research on Parallel Computation, Rice University, December 1994.

[3] J.R. Allen and K. Kennedy. Automatic Translations of Fortran Programs to Vector Form. *ACM Toplas*, 9:491–542, 1987.

[4] P. Amarasinghe, J.M. Anderson, M.S. Lam, and C.-W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[5] S. Andel, B.M. Chapman, J. Hulman, and H.P. Zima. An Expert Advisor for Parallel Programming Environments and Its Realization within the Framework of the Vienna Fortran Compilation System. Technical report, Institute for Software Technology and Parallel Systems, Vienna, Austria, 1996.

[6] C.C Ashcraft, S.C. Eisenstat, J. Liu, and A.H. Sherman. A Comparison of Three Column-Based Distributed Sparse Factorization Schemes. Technical Report YALEU/DCS/RR-810, Computer Science Department, Yale University, 1990.

[7] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges-IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Transactions on Computers*, 28(10):37–47, October 1995.

[8] 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.

[9] 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.

[10] A. Bik and H. Wijshoff. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. Technical Report 92-25, Dept. of Computer Science, Leiden University, 1992.

[11] 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.

[12] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S.X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3):61–74, 1993.

[13] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-Ultimate Tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[14] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, F. Desprez, J.C. Mignot, 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. In J.J. Dongarra and B. Tourancheau, editors, *Third Workshop on Environments and Tools for Parallel Scientific Computing*, Faverges, August 1996. SIAM.

[15] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, F. Desprez, J.C. Mignot, 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 Extensions for Irregular Problems. In J.J. Dongarra and B. Tourancheau, editors, *Third Workshop on Environments and Tools for Parallel Scientific Computing*, Faverges, August 1996. SIAM.

[16] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 459–462. Springer Verlag, August 1996.

[17] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. Technical report, LIP - ENS Lyon, 1996.

[18] T. Brandes and D. Greco. Realization of an HPF interface to ScaLAPACK with Redistributions. In H. Liddel, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking (HPCN)*, volume 1067 of *Lectures Notes in Computer Science*, pages 834–839. Springer Verlag, 1996.

[19] Th. 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, April 1994.

[20] B.M. Chapman, T. Fahringer, and H.P. Zima. Automatic Support for Data Distribution on Distribution on Distributed Memory Multiprocessor Systems. Technical Report TR 93-2, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, August 1993.

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

[22] C. Clémançon, A. Endo J. Fritsher, A. Müller, R. Rühl, and B.J.N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In *28th Hawaii International Conference on System Sciences (HICSS-28)*, volume II, pages 242–251. IEEE Computer Society Press, January 1995.

[23] Alain Darte and Frédéric Vivien. On the Optimality of Allen and Kennedy's Algorithm for Parallelism Extraction in Nested Loops. In *Proceedings of Europar'96*, Lyon, France, August 1996. Springer Verlag. To appear.

[24] Alain Darte and Frédéric Vivien. Optimal Fine and Medium Grain Parallelism in Polyhedral Reduced Dependence Graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press. To appear.

[25] J.-L Dekeyser and C. Lefevre. HPF-Builder: A Visual Environment to Transform Fortran 90 Codes to HPF. In J.J. Dongarra and B. Tourancheau, editors, *Third Workshop on Environments and Tools for Parallel Scientific Computing*, Faverges, August 1996. SIAM.

[26] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In *IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 365–371. Birkhaeuser Verlag AG, Basel, Switzerland, 1994.

[27] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.

[28] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. Technical Report 92-44, LIP - ENS Lyon, December 1992.

[29] A. George and J. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[30] Steven T. Hackstadt and Allen D. Malony. Data distribution visualization (ddv) for performance visualization. Technical report, University of Oregon, Dept. of Computer and Information Science, October 1993.

[31] Steven T. Hackstadt and Allen D. Malony. Distributed array query and visualization for high performance fortran. In *Proc. of Euro-Par '96*, Lyon, France, August 1996.

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

[33] M.T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[34] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0.α.3, Department of Computer Science, Rice University, August 1996.

[35] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. Design and Implementation of the D Editor. In J.J. Dongarra and B. Tourancheau, editors, *Second Workshop on Environments and Tools for Parallel and Scientific Computing*, pages 1–10, Townsend, TN, May 1994. SIAM.

[36] 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 Performanc e Computing Conference*, pages 119–127, Toronto, Ontario, June 1994. ftp://ftp.crim.ca/apar/public/Papers/1994/SS94-EPPP.ps.gz.

[37] C.S. Ierotheou, S.P. Johnson, M. Cross, and P.F. Leggett. Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes. *Parallel Computing*, 22:163–195, 1996.

[38] J.Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, The Universtity of Tennessee - Knoxville, 1995.

[39] J.Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley. LAPACK Working Note: A Proposal for a Set of Parallel Linear Algebra Subprograms. Technical Report 100, The Universtity of Tennessee - Knoxville, 1995.

[40] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New User Iinterface for Petit and Other Extensions.* CS Dept, University of Maryland, April 1996. http://www.cs.umd.edu/projects/omega.

[41] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Library - Version 1.0 - Interface Guide.* CS Dept, University of Maryland, April 1996. http://www.cs.umd.edu/projects/omega.

[42] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Interactive Parallel Programming Using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[43] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[44] P.F. Leggett, A.T.J. Marsh, S.P. Johnson, and M. Cross. Integrating User Knowledge with Information from Parallelization Tools to Facilitate the Automatic Generation of Efficient Parallel FORTRAN Code. *Parallel Computing*, 22:259–288, 1996.

[45] P.A.R. Lorenzo, A. Muller, Y. Murakami, and B.J.N. Wylie. High Performance Fortran Interfacing to ScaLAPACK. Technical Report TR-96-13, Swiss Center for Scientific Computing (CSCS), Manno, Switzerland, 1996.

[46] 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.

[47] 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.

[48] J.-L. Pazat. Tools for High Performance Fortran: A Survey. Technical report, IRISA, Rennes, France, 1996. http://www.irisa.fr/pampa/HPF/survey.html.

[49] A.D. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz. An Overview of the Pablo Performance Analysis Environment. Department of Computer Science, University of Illinois, November 1992.

[50] E. Rothberg and A. Gupta. An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'93*. IEEE Computer Society, 1993.

[51] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'94*. IEEE Computer Society, 1994.

[52] Y. Saad. SPARSKIT: a Basic Tool Kit for Sparse Matrix Computations. Technical Report, Version 2, CSRD, University of Illinois, June 1994.

[53] B.S. Siegel and P.A. Steenkiste. Controlling Application Grain Size on a Network of Workstations. In *Supercomputing'95*, 1995. http://www.cs.cmu.edu/afs/cs/project/nectar/WWW/gnectar_papers.html.

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

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

[56] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[57] M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima. Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. Technical Report TR 95-5, University of Vienna, University of Malaga, 1995.

[58] Springer Verlag, editor. *Proceedings of HPCN Europe 1996*, volume 1067 of *LNCS*, 1996.

[59] M.E. Wolf and M.S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[60] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. Technical report, Institute for Statitics and Computer Science, Vienna, Austria, May 1994.

Contents

List of Figures