Department of Electrónica y Sistemas
University of La Coruña, Spain

PhD THESIS

Parallelization and Compilation Issues of Sparse QR Algorithms

Juan Touriño Domínguez

La Coruña, March 1998

Dr. Ramón Doallo Biempica, Profesor Titular de Universidad del Dpto. de Electrónica y Sistemas de la Universidad de La Coruña.

Dr. Emilio López Zapata, Catedrático de Universidad del Dpto. de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "*Parallelization and Compilation Issues of Sparse QR Algorithms*", ha sido realizada por D. Juan Touriño Domínguez bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidad de La Coruña y concluye la Tesis que presenta para optar al grado de Doctor en Informática con la mención de Doctor europeo.

La Coruña, 15 de Enero de 1998

Fdo: Dr. Ramón Doallo Biempica
Codirector de la Tesis Doctoral

Fdo: Dr. Emilio López Zapata
Codirector de la Tesis Doctoral

Fdo: D. José Mª Domínguez Legaspi
Director del Dpto. de Electrónica y Sistemas

To my parents

Contents

Preface **xvii**

I Parallelization of Sparse QR Algorithms **1**

1 Introduction **3**

1.1 Scope and Motivation 3
1.2 Multiprocessor Architectures 4
1.3 Multiprocessor Programming Environments 6
1.3.1 Message-Passing Programming Model 6
1.3.2 Data-Parallel Paradigm 8
1.3.3 Automatic Parallelizing Compilers 9
1.4 Numerical Libraries 10

2 Dense and Sparse Sequential QR Algorithms **13**

2.1 QR Algorithms 13
2.2 Modified Gram-Schmidt Algorithm 14
2.3 Householder Reflections 16
2.4 Givens Rotations 18
2.5 QR Applications: Linear Least Squares Problems 19
2.6 A Sparse Approach: Data Storage Structures 21
2.7 A Strategy to Preserve Sparsity 24
2.7.1 A New Column Pivoting Criterion 24
2.7.2 Numerical Analysis 29

3 Parallel QR Algorithms **35**

3.1 Data Distribution 35

3.2 Parallel MGS . . . 38
3.3 Parallel Householder Reflections . . . 43
3.4 Parallel Givens Rotations . . . 44
3.5 Parallel Least Squares . . . 46

4 Experimental Results **51**
4.1 Parallel Machines . . . 51
4.2 Execution Times and Efficiencies . . . 53
4.3 Fill-in and Numerical Results . . . 58
4.4 Local Pivoting . . . 62
4.5 Sparse QR Factorization on a Vector Processor . . . 63

II Compilation Issues of Sparse Factorizations **67**

5 A Parallel Library for Sparse Computations **69**
5.1 An Overview of the Library . . . 69
5.2 Library Data Structures and Distributions . . . 70
5.2.1 Data Schemes . . . 70
5.2.2 Data Reorientation . . . 72
5.3 Mapping Iterations . . . 73
5.4 Library Routines . . . 75
5.4.1 Replication Operations . . . 75
5.4.2 Gather Operations . . . 76
5.4.3 Reduction Routines . . . 76
5.4.4 Fill-in Routines . . . 77
5.4.5 Swapping Operation . . . 78
5.4.6 Other Routines . . . 79
5.5 Practical Examples . . . 79
5.5.1 Sparse Matrix-Sparse Matrix Sum . . . 80
5.5.2 Sparse QR Factorizations . . . 81
5.5.3 Sparse Least Squares Problems . . . 84
5.6 Low-Level Features . . . 87
5.6.1 Mesh Functions . . . 87
5.6.2 List Management . . . 88

5.6.3 User-Defined Functions 89
5.7 Comparison Results 89

6 HPF Extensions for Sparse Operations 93
6.1 Introduction 93
6.2 A First Data-Parallel Implementation Using Craft 94
6.3 The SPARSE Directive 96
6.4 HPF-2 Proposals for Sparse Distributions 97
6.5 Extended HPF-2 Sparse Codes 101
6.5.1 Sparse MGS Data-Parallel Code 102
6.5.2 Sparse Householder Data-Parallel Code 108
6.5.3 Least Squares Problems 111

7 Compiler and Run-Time Support 115
7.1 Compiler Support 115
7.1.1 Source Code Analysis 115
7.1.2 Parallelization and Code Generation 119
7.2 Run-Time Support 120
7.2.1 Declaration Section 121
7.2.2 Executable Section 125
7.3 Experimental Results 127
7.3.1 Parallel MGS Generated Code 127
7.3.2 Parallel Householder Generated Code 129
7.4 Comparison with the CHAOS Approach 131
7.5 Towards Automatic Parallelization 132
7.5.1 The SUIF and Polaris Compilers 132
7.5.2 Automatic Parallelization of Sparse Computations 133
7.5.3 Testing Monotonicity of Index Arrays 136
7.5.4 Testing Non-Overlapping Ranges of Induction Variables . . 137

Conclusions 141

Bibliography 145

List of Tables

2.1 *Harwell-Boeing sparse matrices* 26

4.1 *Characteristics of the target supercomputers* 53

4.2 *MGS: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)* 54

4.3 *Householder: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)* . . 54

4.4 *Givens: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists* 54

4.5 *MGS: execution times and efficiencies on the Cray T3D for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)* 55

4.6 *Householder: execution times and efficiencies on the Cray T3D for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)* . . 55

4.7 *Givens: execution times and efficiencies on the Cray T3D for* $\epsilon = 1$*, using doubly-linked lists* 55

4.8 *Number of nonzero entries in matrix R using several mesh configurations (*$\epsilon = 1$*)* 59

4.9 *Numerical errors using several mesh configurations (*$\epsilon = 1$*)* 59

4.10 *Global pivoting vs local pivoting* 63

4.11 *Execution times on a vector processor* 65

5.1 *MGS: execution times and speed-ups for matrices JPWH991, ORANI678 and SHERMAN5* 90

7.1 *Benchmark for list management in Fortran 90 and C* 129

7.2 *Execution times for the parallel sparse MGS using manual LLCS and the CHAOS implementation* 131

List of Figures

2.1 *MGS algorithm* 14
2.2 *Column swap (pivoting)* 15
2.3 *Process of updating matrices Q and R* 15
2.4 *Householder algorithm* 17
2.5 *Process of obtaining matrix R from the original matrix A* 17
2.6 *Pseudo-code for Givens rotations* 18
2.7 *Sequential Givens rotations* 19
2.8 *Data structures for sparse matrices: doubly-linked lists and packed vectors* 22
2.9 *Patterns of the Harwell-Boeing test matrices* 26
2.10 *MGS: fill-in in matrix R for different values of* ϵ 27
2.11 *Householder: fill-in in matrix R for different values of* ϵ 28
2.12 *Givens: fill-in in matrix R for different values of* ϵ 28
2.13 *SHL400: numerical errors for different values of* ϵ 31
2.14 *JPWH991: numerical errors for different values of* ϵ 31
2.15 *SHERMAN1: numerical errors for different values of* ϵ 32
2.16 *MAHINDAS: numerical errors for different values of* ϵ 32
2.17 *ORANI678: numerical errors for different values of* ϵ 33
2.18 *SHERMAN5: numerical errors for different values of* ϵ 33

3.1 *Load balancing using a cyclic distribution for Householder reflections* 36
3.2 *Sparse matrix data distribution scheme* 37
3.3 *Pivoting: buffer for sending local lists* 39
3.4 *Broadcast and reduction in the parallel MGS procedure* 41
3.5 *Fill-in in the MGS algorithm* 42
3.6 *Row access in a column-oriented data structure* 45
3.7 *Parallel Givens rotations* 46

3.8 *Calculation of $Q^T b$* 47

4.1 *Architecture of the Fujitsu AP1000* 52

4.2 *Architecture of the Cray T3D* 52

4.3 *Matrix JPWH991: execution times for the QR algorithms on the Cray T3D* 56

4.4 *Matrix ORANI678: execution times for the QR algorithms on the Cray T3D* 57

4.5 *Matrix SHERMAN5: execution times for the QR algorithms on the Cray T3D* 57

4.6 *Matrix JPWH991: fill-in in matrix R for different mesh sizes* . . . 60

4.7 *Matrix ORANI678: fill-in in matrix R for different mesh sizes* . . 61

4.8 *Matrix SHERMAN5: fill-in in matrix R for different mesh sizes* . 61

4.9 *Architecture of the Fujitsu VP2400/10 vector computer* 64

4.10 *Sparse code examples for a vector processor* 65

5.1 *LLCD scheme* 71

5.2 *List reorientation, from column-direction to row-direction* 72

5.3 *Vector redistribution, from column-direction to row-direction* . . . 73

5.4 *Mapping global loops onto local loops* 74

5.5 *An example of iteration space* 74

5.6 *Singly-linked list data structure* 75

5.7 *Replication of one column of a matrix* 76

5.8 *Gather operation* 77

5.9 *Column-oriented sparse matrix-sparse matrix sum* 80

5.10 *Row-oriented sparse matrix-sparse matrix sum* 80

5.11 *Sparse MGS code using 3LM routines* 81

5.12 *Code to preserve sparsity in the MGS algorithm* 82

5.13 *Sparse Householder code using 3LM routines* 83

5.14 *Row-oriented Householder transformations* 84

5.15 *Row-oriented sparse upper triangularization using Householder transformations (3LM code)* 85

5.16 *3LM back-substitution* 86

5.17 *Permutation of the entries of the solution vector x* 87

5.18 *Data structures for list management* 88

5.19 *3LM low-level code* 89

6.1 *Sparse QR Craft example codes* 95
6.2 *HPF specification of the BRS distribution* 96
6.3 *BRS partitioning for a 2×2 processor mesh* 97
6.4 *LLRCS data storage scheme* 98
6.5 *Fortran 90 derived data types for the list items and declaration of an array of pointers to these items* 99
6.6 *Syntax for the proposed HPF-2 SPARSE directive* 100
6.7 *HPF-2 specification of a replicated compressed vector* 101
6.8 *Alignment and distribution of a sparse array on a 2×2 mesh* . . . 102
6.9 *Declaration section of the extended HPF-2 specification of the MGS algorithm* 103
6.10 *Partitioning of MGS arrays on two processors* 104
6.11 *Outline of an extended HPF-2 specification of the parallel MGS algorithm* 105
6.12 *Fortran 90 module for list management* 107
6.13 *Declaration section of the extended HPF-2 specification of the Householder algorithm* 108
6.14 *Outline of an extended HPF-2 specification of the parallel Householder algorithm* 109
6.15 *Extended declaration section for the LSP through MGS* 111
6.16 *HPF-2 back-substitution for the LSP* 112

7.1 *Scanner specification for the SPARSE directive* 116
7.2 *Parser specification for the SPARSE directive* 117
7.3 *Output of the HPF-2 compiler for the declaration section of the MGS algorithm* 122
7.4 *Output of the HPF-2 compiler for the executable section of the MGS algorithm* 123
7.5 *DDLY functions for mapping loop iterations* 125
7.6 *MGS: exec. times using F90 linked lists and Cray T3E SHMEM* . 128
7.7 *MGS: speed-ups using F90 linked lists and Cray T3E SHMEM* . . 128
7.8 *Householder: execution times using F90 linked lists and Cray T3E SHMEM* 130
7.9 *Householder: speed-ups using F90 linked lists and Cray SHMEM* . 130
7.10 *Sparse matrix-dense vector product* 133

7.11 *Summarized output of the SUIF compiler for the sparse matrix-dense vector product* 134

7.12 *Summarized output of the Polaris compiler for the sparse matrix-dense vector product* 135

7.13 *Subscripted array subscripts in the QR codes* 136

7.14 *Fill-in stage of the Fortran 77 sparse QR algorithms* 137

7.15 *Test of non-overlapping ranges of the variable bufindex* 139

Preface

The general subject of this thesis is the parallelization of algorithms for sparse matrices, that is, matrices with many zero entries.

To be more specific, we present in this work techniques for parallelizing sparse algorithms, following four different (but interconnected) approaches that depend on the programming efforts of the user. Thus, we first analyze parallelization techniques completely carried out by the programmer; in a second stage, the programmer is assisted by a library of routines to make the development of parallel sparse algorithms easier. Regarding the third approach, the user reduces programming efforts, in a data-parallel programming context, using compiler directives to aid the compiler in generating efficient parallel code (semi-automatic parallelization). Finally, we point out some ideas about fully automatic parallelization techniques for sparse codes.

As a unifying example of all these different approaches, we have focused on sparse QR factorization algorithms, although the ideas presented in this dissertation may be applied, without losing generality, to a wide variety of sparse algorithms.

The thesis is composed of two parts. In the first part (Chapters 1-4) we discuss the manual parallelization of sparse QR algorithms, paying special attention to sparse data structures and distributions. The second part (Chapters 5-7) takes advantage of the user parallelization experiences obtained in the first part and tries to incorporate those techniques into a parallel library and into a parallel compiler, with the purpose of releasing the user from annoying parallelization tasks.

The first chapter is a brief description of the context of the thesis, including multiprocessor architectures and their corresponding programming environments.

Chapter 2 introduces QR factorization algorithms: Modified Gram-Schmidt, Householder reflections and Givens rotations, with application to least squares problems. Data storage structures for the sparse versions of these algorithms and a strategy to reduce the fill-in during the factorizations are also described.

Chapter 3 discusses the techniques used to parallelize the factorizations described in Chapter 2 and shows data distribution schemes suitable for sparse algorithms.

Chapter 4 presents an experimental evaluation on the target machines (dis-

tributed memory multiprocessors) of the parallel algorithms described in Chapter 3, focusing on execution times, efficiencies, numerical accuracies and fill-in results. Experiments on a vector processor were also included.

Chapter 5 details the routines of the parallel library we named *3LM*, whose aim is to facilitate the programming of parallel sparse algorithms, specially factorizations with pivoting operations and fill-in. This chapter begins the second part of the thesis and may be considered as a transitional and linking chapter between the two parts of the dissertation.

Chapter 6 proposes the extension of the syntax of a data-parallel language (*High Performance Fortran*) to incorporate data structures and distributions suitable for sparse matrix computations.

Chapter 7 covers all the compiler and run-time technical features necessary to support the data-parallel extensions proposed in Chapter 6. The good behaviour of these language extensions to generate efficient parallel code is justified experimentally. A brief overview about automatic parallelization of sparse codes is also included.

The results of this research work have been published in [7] [28] [29] [30] [31] [37] [94] [95] [97] and [98] (spanish conferences and technical reports were not included).

Acknowledgments

The individual effort of the writer is a compulsory condition in order to successfully write a thesis, but it is not enough. The success of a thesis also depends, to a large extent, on the PhD advisors. I was very lucky, in this respect, to have two excellent directors: *Dr. Ramón Doallo* (University of La Coruña) and *Dr. Emilio L. Zapata* (University of Málaga). Their rigorous supervision and constant support have been decisive and to them I owe a great deal of gratitude.

I also want to acknowledge *Rafael Asenjo* for his invaluable collaboration in some chapters of this work, and for his friendship. *Basilio B. Fraguela* also deserves to be mentioned because of his aid in some technical aspects. Thanks to *María J. Martín* for providing me with the SUIF codes of Chapter 7 during her research visit at the Stanford University.

I gratefully thank to the following institutions for funding this work:

- *Department of Electrónica y Sistemas of La Coruña*, for the human and material support.

- *University of La Coruña*, for financing my attendance at some conferences and my visits to the *University of Edinburgh* and to the *University of Minho.*

- *Xunta de Galicia*, for the projects XUGA10501A92 (Diseño de un Procesador/Codificador Basado en la Técnica de Estimación del Movimiento)

and XUGA20605B96 (Diseño de Algoritmos Numéricos Irregulares sobre Arquitecturas Masivamente Paralelas).

- *Ministry of Education of Spain*, for the projects CICYT TIC95-1754-E (Computación Paralela: Diseño de Arquitecturas VLSI) and TIC96-1125-C03 (Diseño de Arquitecturas y Paralelizadores en Computación Masivamente Paralela: Aplicaciones Dinámicas y/o Basadas en Matrices Dispersas).

- *European Union*, for the TRACS Programme (Training and Research on Advanced Computing Systems) of the Human Capital and Mobility Programme, grant ERB-CHGE-CT92-0005.

I acknowledge the following supercomputing centres for giving me remote access to their machines:

- *Fujitsu High Performance Computing Research Centre*, FPCRF, Kawasaki, Japan (Fujitsu AP1000).

- *Imperial College/Fujitsu Parallel Computing Research Centre*, IFPC, London, UK (Fujitsu AP1000).

- *Edinburgh Parallel Computing Centre*, EPCC, Edinburgh, UK (Cray T3D).

- *Ecole Polytechnique Federale de Laussane*, EPFL, Switzerland (Cray T3D).

- *Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas*, CIEMAT, Madrid, Spain (Cray T3E).

- *Centro de Supercomputación de Galicia*, CESGA, Santiago de Compostela, Spain (Fujitsu VP2400/10).

Finally, I also thank to the *Department of Arquitectura de Computadores* (University of Málaga, Spain), the *EPCC*, the *Department of Mathematics and Statistics* (University of Edinburgh, UK) and the *Department of Matemática* (University of Minho, Braga, Portugal) for their assistance during my research visits.

Juan Touriño

Part I

Parallelization of Sparse QR Algorithms

Chapter 1

Introduction

1.1 Scope and Motivation

Supercomputing has become an essential tool for scientific and industrial research. The application of high-performance computing for numerically intensive problems brings several new issues that do not appear in standard computing. The process of mapping or transforming abstract problems into concrete solutions that execute rapidly on high-performance computers is a difficult task. New algorithms, programming tools and compilers are required to effectively exploit the power of supercomputers.

Among the great variety of mathematical areas suitable for high-performance computing, linear algebra is an area of considerable interest because it is the computational core of many computational scientific and engineering problems; for instance, several applications require solving large systems of linear equations. As the scope of linear algebra is very broad, this thesis focuses on parallelization techniques for matrix factorization algorithms and, more specifically, for the QR factorization (with application to linear least squares problems), although many of the fundamental ideas expressed in the dissertation can be applied to other kinds of problems.

Many codes for dense factorizations on parallel computers have been proposed and analyzed. But sparse factorizations occur in a variety of applications. Therefore, our scope is even more specific: we consider that the matrix to be factorized has many null entries, which requires a special treatment, as well as specific parallelization techniques. This way, significant gains in execution time and memory usage, sometimes decisive for the possibility of solving large problems, can be made.

In order to delimit the context of the thesis, in this introductory chapter we describe, without getting bogged down in details, current multiprocessor architectures, as well as the programming models for writing parallel codes in these machines. An overview of linear algebra libraries for dense and sparse computations is also provided.

1.2 Multiprocessor Architectures

The increasing demand of memory and CPU resources by computational applications has caused the development of faster architectures for high-performance computing.

The technological advances in processor design has led to the exploitation of instruction-level parallelism, which reduces the mean number of clock cycles to execute an instruction or that execute several instructions in parallel, using several independent functional units. Examples of this kind of parallelism are the superscalar and VLIW (Very Long Instruction Word) architectures. The performance of this kind of processors is strongly influenced by the ability of the compiler to restructure the machine code to maximize the number of functional units working in parallel. However, the increase in the performance of a processor due to instruction-level parallelism is limited; for instance, nowadays a processor architecture that executes eight or more instructions in parallel is not profitable. Besides, the real performance is very far from the theoretical one due to the characteristics of the codes and the use of inefficient compilers.

Another approach is to exploit a coarser grain parallelism. This fact justifies the relevance of multiprocessors, computers with a set of processors which work together on the solution of complex computational problems (mainly *Grand Challenge* scientific applications). When they are made up of a large number of processors, the denomination of MPP (Massively Parallel Processor) systems is employed.

Detailed and updated information about multiprocessors can be found in the following recent computer architecture bibliography: [27][54, Chapter 8][64] and [89].

Basically, multiprocessors can be classified into two main groups:

- *Shared memory multiprocessors.* In this model, there is a global memory and all the processors can have direct access to all the memory positions (global address space); consequently, the programming model of these machines is not complex. Data written in memory by one processor are accessible by any other processor; therefore, the communication among processors takes place through the global memory. The main drawback of this architecture is the lack of scalability because the common memory only supports the connection of a reduced number of processors. Examples of shared memory machines are the SGI Power Challenge, Cray Y-MP, Cray C-90.

- *Distributed memory multiprocessors.* The processors have local memories and the communication and synchronization operations are performed through message-passing routines. Therefore, the access to non-local data is highly penalized compared to a local access. These machines are more difficult to program than the shared memory computers, but in contrast they present a good scalability. Well-known examples of this kind of supercomputers are the IBM SP2, CM-5 (Connection Machine), Fujitsu AP1000/3000,

Intel Paragon.

The current trend in multiprocessors leads to an architectural concurrence which tries to join the advantages of both models: scalability and ease of programming. As a result, we have the so-called *distributed-shared* memory architectures, in which the memory is physically distributed (distributed memory), but there is a software or a software/hardware support that allows a global address space (shared memory) and, in some cases, cache coherence management. That is, they are distributed memory machines, but they can virtually be programmed as shared memory machines, in a transparent way for users.

The distributed-shared memory supercomputers can be classified into two groups:

- *NUMA (Non-Uniform Memory Access)* machines or multiprocessors with a static physical address space. We find different architectures in this group depending on the cache coherence mechanism:
  - Without cache coherence: for instance, the Cray T3D.
  - With partial cache coherence: Cray T3E.
  - With cache coherence (CC-NUMA, Cache Coherent NUMA): SGI Origin 2000.
- *COMA (Cache Only Memory Access)* machines or supercomputers with a dynamic physical address space, where the local memories are converted into caches. This is a particular case of a NUMA architecture in which there is no memory hierarchy in each processor. An example is the KSR-1 (Kendall Square Research).

As can be observed, we have used the term multiprocessor only for MIMD (Multiple Instruction Multiple Data) architectures, where the processors can execute programs independently. We have not considered SIMD (Single Instruction Multiple Data) supercomputers, also known as array processors, in which each instruction is executed, in a synchronous way, on all the processors. Examples of SIMD machines are the CM-2 (Connection Machine) and the MasPar MP-2.

A network of computers can be considered a distributed memory multiprocessor and it represents a cheap and attractive alternative for those research groups that cannot have access to supercomputers. This cluster can also be used to debug parallel programs (programmed using portable message-passing libraries) before porting them to a supercomputer.

The Fujitsu AP1000, Cray T3D and Cray T3E multiprocessors and a cluster of workstations were the target machines used in our experiments, as will be shown in Chapters 4, 5 and 7.

1.3 Multiprocessor Programming Environments

A multiprocessor is programmed in order to exploit parallelism, dividing the problem into the available processors. The parallelism can be obtained at several levels according to the granularity:

- *Instruction-level (fine grain)*, as we mentioned in the previous section, for the superscalar and VLIW processors (obviously, this kind of parallelism is at the microprocessor level).
- *Loop-level (medium grain)*, in which the iteration space of the program loops without dependences is distributed among the different processors. Numerical applications are good candidates for exploiting loop-level parallelism.
- *Task-level (coarse grain)*, which can be applied when there are semi-independent tasks that can be executed simultaneosly on several processors. The simplicity of this model is very adequate for operating systems, such as Solaris, Windows NT or OS/2, to exploit this kind of parallelism on a shared memory machine with few processors. The communications among these tasks or threads (multithread model) takes place through the shared memory.

In this thesis, we well focus on the second kind: the extraction of loop-level parallelism in our algorithms. In order to achieve this goal, we will describe in the next subsections the three programming paradigms, in descending order of programming difficulty and flexibility, we have used to program our target machines: message-passing programming model, data-parallel paradigm and automatic parallelizing compilers.

Although in an MIMD environment each processor could have a different program running, we have used for all our algorithms the SPMD (Single Program Multiple Data) programming paradigm. Under this model each processor runs the same program but executes different code depending on its processor identifier and the data held in its local memory. It is, basically, an SIMD style on an MIMD computer.

1.3.1 Message-Passing Programming Model

The message-passing environment provides the primary mechanism for programming multiprocessor applications, tipically MIMD distributed memory multiprocessors and networks of computers (even heterogeneous systems, composed of different processor architectures).

Message-passing provides read/write access for each processor to the local memory of all the other processors (no other access is allowed) and synchronization of processes. The main advantage is the high flexibility and the absolute

control of the program, although it also represents the main drawback because the user is responsible for data and computation distribution on the processors, communications, synchronizations and program optimizations (for instance, to minimize communications by exploiting data locality using the owner-computes paradigm, in which all computations updating a given datum are performed by the processor owning that datum). Therefore, this programming model has a high cost of development and debugging; to make a comparison, writing message-passing codes in a multiprocessor can be compared with the assembly programming of a monoprocessor.

Initially, message-passing libraries were only machine specific and, therefore, incompatible. Examples of these libraries are CROS (Crystalline Operating System) of the Caltech Hypercube; NX1/NX2, to program the Intel iPSC family; PSE (Parallel Software Environment), a set of communication primitives of the nCUBE; EUI (External User Interface), the message-passing system for the IBM SP MPP computer series; Thinking Machines CMMD, for the CM-5, etc., as well as the native communication calls of the Fujitsu AP1000 [38], which were used to program our algorithms (see Section 4.1). We have also employed the SHMEM [11] native low-level libraries of the Cray T3D/T3E in the experimental results obtained in Section 7.3.

As the services provided by the message-passing libraries are basically the same, several groups (usually vendor independent) have tried to overcome the machine specificity of the message-passing libraries by developing portable message-passing environments. Examples of such efforts are the portable libraries EXPRESS, PARMACS, Zipcode and mainly PVM (Parallel Virtual Machine) [40] and MPI (Message Passing Interface) [71], which are the two portable message-passing libraries we have used in our codes (PVM codes in Sections 4.1, 5.1 and 7.4 and an MPI implementation in 7.3).

PVM and MPI are composed of the C or Fortran 77 uniprocessor programming languages, along with message-passing routines. MPI is prevailing as message-passing standard, defined by the MPI Forum, a committee composed of vendors, universities and research laboratories in the field of parallel computing. MPI intends to include the major features of all vendor systems in an efficient way. Although MPI includes a complete set of message-passing routines, in our opinion it is oversized (mainly after the MPI-2 Forum [72]). PVM, developed by the Oak Ridge National Laboratory, was initially conceived to allow a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource and, afterwards, efficient implementations for programming supercomputers were designed. PVM does not provide as many features as MPI, but its generality and its simplicity of design makes PVM be widely used.

McBryan [69] presents an interesting and complete review of native and portable message-passing interfaces, including references about the libraries mentioned in this subsection.

1.3.2 Data-Parallel Paradigm

The data-parallel programming model, which inherits the SIMD style, arises principally to simplify the programming of distributed memory message-passing systems. In this approach, the programmer writes a sequential program (that is, using a global address space) in a standard language (C, Fortran 77, Fortran 90) and guides the compiler in its work of generating parallel code by means of a set of compiler directives or annotations to indicate, for instance, the appropriate data distribution for a specific program. The data array and computation distribution (loop-level parallelism) is the core of this paradigm.

Basically, for an MIMD multiprocessor architecture (including workstation networks), a data-parallel compiler transforms the program into an SPMD code by partitioning and distributing its data as specified, allocating computation to processors according to the locality of the data references involved, and inserting any necessary data communications by message-passing or by a shared memory mechanism at appropriate points in the program. Besides, in a distributed memory architecture, the compiler is in charge of the tedious low-level details of translating from an abstract global address space to the local memories of individual processors. As accesses to local memory data are much faster than non-local accesses, it is important, for efficiency reasons, that users choose (by means of compiler directives) the adequate data and computation partitions in a way that the compiler minimizes communications and maximizes data parallelism; that is, the programmers will need a good understanding of the code and of the meaning of the data-parallel directives to obtain efficient codes.

Although some control of the program is lost and the efficiency of the generated parallel code is, in general, not as good as the manual parallel codes programmed using the message-passing model, the data-parallel codes are shorter, clearer and easier to develop and modify than their message-passing equivalents. Besides, it is easier to experiment with different data distributions by simply changing the corresponding directives in the data-parallel code than by recoding a message-passing program.

Nevertheless, in some occasions, the parallel code generated by a data-parallel compiler can be better than the manual one because the implementation of the communications in a data-parallel compiler is usually based on native machine instructions (for instance, the Craft compiler of the Cray T3D/T3E, uses SHMEM routines) instead of slower message-passing portable libraries (PVM, MPI), employed in the manual codes.

Well-known data-parallel languages are CM Fortran (Connection Machine Fortran) [93], Vienna Fortran [110], Fortran D [36], Craft [76] and, specially, the standard High-Performance Fortran (HPF) [56][57], which allows writing portable data-parallel programs across a wide variety of parallel machines. We will discuss the data-parallel paradigm again in Chapter 6, where we propose the inclusion of directives for processing irregular codes (more specifically, sparse matrix codes) in the HPF compiler.

1.3.3 Automatic Parallelizing Compilers

This is the simplest programming model for users, as the compiler automatically generates the equivalent parallel version of the sequential code written in a conventional language (C, and particularly Fortran 77, the dominant language in high-performance computation), so that multiprocessor parallelism is transparent to programmers. These compilers mainly focus on the transformation (automatic restructuring) of numerical programs for large-scale scientific and engineering applications. However, this is a very complex task, specially to automatically determine a suitable data and computation distribution, because it requires a global analysis of the program's data access patterns and often this information cannot be determined statically.

The books of Zima [111] and Wolfe [106] are interesting bibliography about techniques associated with automatic code restructurers (for instance, data dependence analysis to determine if the statements of a loop can be executed concurrently). Wolfe also focuses on techniques to generate optimized parallel code depending on the target architecture.

A parallelizing compiler can have success for certain well-known access patterns, but real codes present complex loops, calls to procedures, complicated data structures, etc. which make these compilers fail. Besides, the compiler does not usually have the semantic knowledge of the user about the behaviour of the program and, therefore, the output parallel code generated is, in general, very inefficient.

Outstanding pioneer restructuring tools to analyze and transform sequential code into parallel code are Parafrase, developed at the University of Illinois at Urbana-Champaign (the current version is Parafrase-2 [77]), and SUPERB (Bonn University) [109], although this last one is, in fact, a semi-automatic parallelization system. More recent restructurers are SUIF (developed at the Stanford University) [50] and Polaris (University of Illinois at U.C.) [16], which are further detailed in Subsection 7.5.1.

Nowadays, automatic parallelization is still the domain of universities and research institutions. Many research efforts are being carried out in the development of new compiling techniques for parallelizing compilers, which is an active research area, and much work has to be done to obtain satisfactory results with real numerical scientific codes and thus, to bring into general use this kind of compilers to program supercomputers efficiently. The semi-automatic approach provided by data-parallel compilers (explained in the previous subsection), which are aided by the programmer using directives, is the cost users must pay to make the efficient parallel code generation easier in relation to fully automatic compilers.

In general, code restructuring works properly when the codes to be parallelized have regular data access patterns, but the automatic parallelization of irregular codes is much more complicated. We will describe automatic parallelization techniques for our sparse codes in Section 7.5.

1.4 Numerical Libraries

As the general application scope of this thesis is linear algebra problems, it would be interesting to review existing numerical libraries for linear algebra. They are usually Fortran 77 libraries (although there are C, C++ and Fortran 90 ones). Much of the work in developing portable linear algebra software for advanced architecture computers is motivated by the need to solve large problems on the fastest computers available.

BLAS (Basic Linear Algebra Subprograms) [63] represents the beginning of this kind of libraries. BLAS was originally designed to encapsulate basic vector operations highly tuned and optimized for a given computer architecture (for instance, by means of an adequate management of the memory hierarchy), while the high-level routines that call them remain portable. There are three BLAS sets of routines (or levels): level 1 (vector operations), level 2 (matrix-vector operations) and level 3 (matrix-matrix operations).

Among the wide variety of linear algebra libraries, we must point out LINPACK [32], which is a standard library of Fortran subroutines that analyze and solve systems of linear equations (dense systems and those having special properties, such as symmetric, triangular or banded) and linear least squares problems. These high-level routines are based on lower-level BLAS.

LAPACK (Linear Algebra Package) [4] is a more recent library that updates the entire LINPACK collection and the EISPACK library (routines for eigenvalue and eigenvectors calculation) for higher performance on modern computer architectures (for instance, shared memory computers). It also includes many new capabilities. The LAPACK90 software library is the Fortran 90 version of LAPACK. ScaLAPACK [23] extends the LAPACK library to run scalably on MIMD distributed memory computers, exploiting loop-based parallelism. Following this approach, in Chapter 5, we will describe our *3LM* library, developed to support sparse matrix operations on distributed memory multiprocessors.

LINPACK and LAPACK include routines for calculating the QR factorization, mainly for solving linear least squares problems, which is the application of QR factorization we will focus on (consult Section 2.5). A complete list of numerical software that solves dense least squares problems through QR factorization is presented in [52, Chapter 3].

The software described above are dense linear algebra libraries, not for general sparse matrices, which are our main subject. The books of Zlatev [112] and Duff et al. [33] are useful references to plunge into the sparse linear algebra realm. Many sparse numerical libraries were mainly developed to solve sparse linear systems of equations using iterative methods. The report of Meier et al. [70] cites software for these sparse computations, such as ITPACK, SPARSKIT, NSPCG, RPPACK, etc., as well as the sparse library of the authors, SPLIB.

SPARSPAK (The Waterloo Sparse Matrix Package) is a collection of Fortran subroutines for solving large sparse systems of equations and large sparse least squares problems. HSL (Harwell Subroutine Library) [1] is a wide suite of

Fortran 77 subroutines and Fortran 90 modules for scientific computation which include sparse linear algebra algorithms to solve a great variety of sparse linear systems, sparse least squares, etc. Finally, NIST Sparse BLAS [82] is a C library (although a Fortran version is being developed) which provides, in the same way as BLAS for dense matrices, sparse matrix computational kernels (for instance, sparse matrix or sparse vector products, solution of triangular systems $\cdots$) and supports many storage formats for the sparse matrices.

We can find abundant public domain linear algebra software, both for dense and sparse matrices in software servers such as *netlib* (at http://www.netlib.org) and the guide to available mathematical software of the National Institute of Standards and Technology, NIST (at http://gams.nist.gov).

Chapter 2

Dense and Sparse Sequential QR Algorithms

2.1 QR Algorithms

QR factorization is a direct method in linear algebra based on orthogonalization. It involves the decomposition of a matrix $A \in \Re^{m\times n}$ $(m \geq n)$ into the product of two matrices: $A = QR$, where Q is an orthogonal matrix ($Q^T = Q^{-1}$) and R is upper triangular. The good numerical behaviour of the orthogonal transformations, low sensitivity to rounding errors and good numerical stability are important advantages of the QR factorization when compared to other techniques for the solution of problems in linear algebra. Another important characteristic of QR decomposition is the large variety of applications it has, as will be shown in Section 2.5.

There are several methods for calculating this factorization, compiled in [42, Chapter 5] and [52, Chapter 3]: the Modified Gram-Schmidt procedure (MGS), Householder reflections or transformations and Givens plane rotations. There are also hybrid algorithms, which employ both Householder transformations and Givens rotations in different phases, such as the one described by Pothen and Raghavan in [80]. QR algorithms adapted to certain structured matrices were also developed; for instance, Chun et al. [24] present a family of new fast QR algorithms for Toeplitz matrices [42, Chapter 4].

While LU factorization (Gaussian elimination) is widely used and researched, QR decomposition is used less frequently in applications because it is generally more expensive to compute than the LU factorization. However, there are many applications that produce rank-deficient matrices or that require least squares solutions, for which Gaussian elimination is unsuitable. Moreover, many of these applications generate sparse matrices, a property that can be exploited to reduce the cost of computation.

The QR algorithms we will describe in the next sections are rank-revealing QR factorizations with column pivoting, for both approaches: dense and sparse.

```
rank = n;
for (j=0; j<n; j++)
    norm_j = sum_{i=0}^{m-1} a_{ij}^2;                                    (2.1)

for (k=0; k<n; k++) {
    Find p, k <= p < n, such that norm_p = max_{k<=j<n} norm_j;         (2.2)

    if (norm_p < phi) {                                                  (2.3)
      rank = k;    break;
    }
    swap (norm_k, a_{0:m-1,k}, r_{0:n-1,k}) and
         (norm_p, a_{0:m-1,p}, r_{0:n-1,p});                             (2.4)

    r_{kk} = sqrt(norm_k);                                               (2.5)

    for (i=0; i<m; i++)
      a_{ik} = a_{ik}/r_{kk};                                            (2.6)

    for (j=k+1; j<n; j++) {
      r_{kj} = sum_{i=0}^{m-1} a_{ik} * a_{ij};                          (2.7)
      norm_j = norm_j - r_{kj}^2;                                        (2.8)
      for (i=0; i<m; i++)
        a_{ij} = a_{ij} - a_{ik} * r_{kj};                               (2.9)
    }
}
```

Figure 2.1: *MGS algorithm*

2.2 Modified Gram-Schmidt Algorithm

The MGS algorithm is a rearrangement of the Classical Gram-Schmidt (CGS) algorithm with better numerical properties, because CGS presents a loss of orthogonality among the computed columns of Q. The MGS method obtains matrices $Q \in \Re^{m\times n}$ and $R \in \Re^{n\times n}$.

Matrix A is overwritten by matrix Q (in-place algorithm). Figure 2.1 presents the sequential algorithm with column pivoting in order to consider those situations in which the rank of matrix A is not maximum (rank-deficiency cases) and to provide numerical stability (from now on, for all our C algorithms, we assume that the arrays begin in index 0).

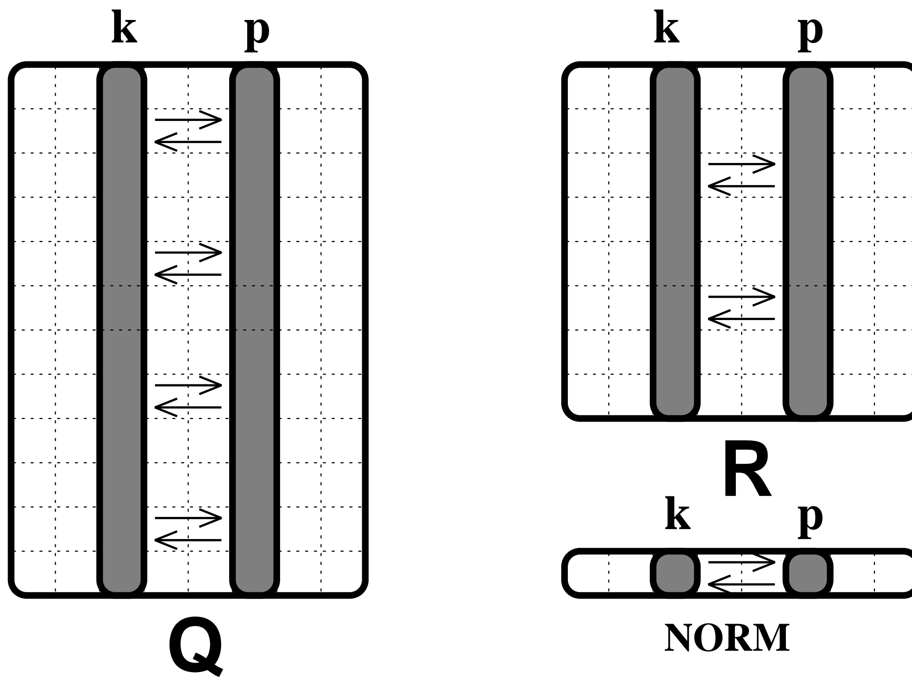

Figure 2.2: *Column swap (pivoting)*

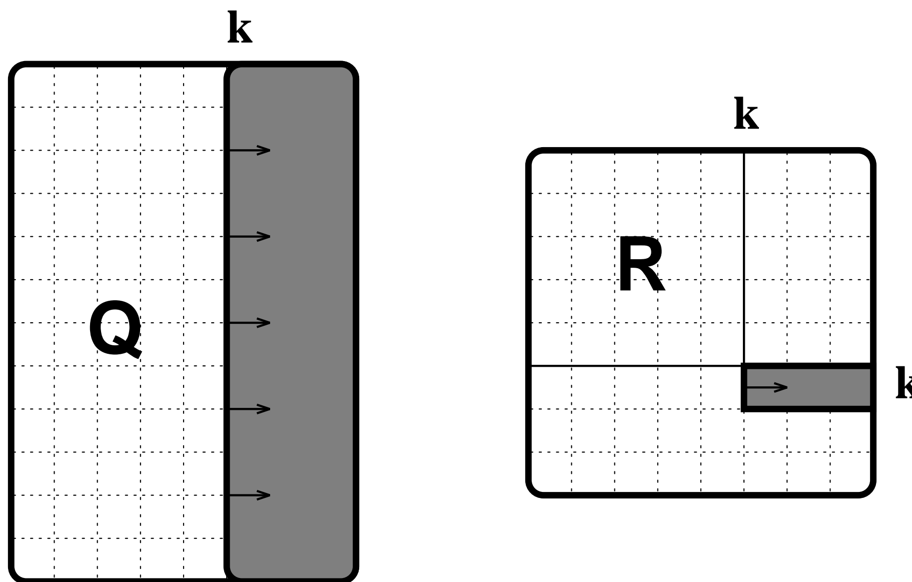

Figure 2.3: *Process of updating matrices Q and R*

In (2.1) the squares of the euclidean norms of the columns of matrix A are calculated and stored in vector *norm*. Then, a loop of n iterations (if the rank of A is maximum) is performed. This consists of the following actions: the pivot column (p) and the pivot element ($norm_p$) are selected (2.2); the pivot element is the maximum of the norms of the columns whose index is $\geq k$. If the pivot is close to 0 (ϕ is the precision required, if $x \in \Re < \phi$, x is considered zero), the rank of the matrix is k and the factorization ends (2.3); otherwise, a swap of column k with the pivot column in matrices Q and R, as well as a swap of their norms are performed (2.4), as shown in Figure 2.2. The use of orthogonal transformations is numerically stable and, in practice, the rank of the matrix can be determined accurately when column permutations are performed during factorization (rank-revealing QR factorization).

In (2.5-2.9) we update matrix A entries $a_{0:m-1,k:n-1}$ (2.6),(2.9) and matrix R entries $r_{k,k:n-1}$ (2.5),(2.7). We avoid undesirable divisions by a value close to 0 in (2.6) thanks to the pivoting. The shaded sections of Figure 2.3 show the entries which are updated in iteration k. The corresponding norms are also updated in expression (2.8).

Once the algorithm has ended, what we really obtain is an $A \times \Pi = Q \times R$ factorization, where Π is a permutation $n \times n$, made up of the product of *rank* elementary permutations: $\Pi = \pi_0 \times \pi_1 \times ... \times \pi_{rank-1}$, each π_j, with $j=0,...,rank-1$, being the identity matrix or a matrix resulting from swapping two of its columns. This is due to the pivoting carried out in (2.4).

2.3 Householder Reflections

In the Householder algorithm, matrix $A \in \Re^{m \times n}$ is overwritten by matrix $R \in \Re^{m \times n}$ (in-place factorization, in which the entries of the last $m - n$ rows are zero), and matrix $Q \in \Re^{m \times m}$ is not explicitly obtained. The Householder algorithm with column pivoting, shown in Figure 2.4, is similar to MGS, substituting in Figure 2.1 expressions (2.5-2.9) by (2.10-2.14). Moreover, only matrix A (which is matrix R once the factorization is completed) and its norms are involved in the column pivoting stage (2.4).

Let us describe the sequential algorithm: in (2.10), the *Householder vector* $v \in \Re^{m-k}$ is calculated, so that the product between $P = (I - 2vv^T/v^T v)$ and the subcolumn $a_{k:m-1,k}$ is zero in all the components except for the first one; I is the identity matrix and P, an orthogonal matrix, is called a *Householder reflection* or *transformation* (P and $I \in \Re^{(m-k)\times(m-k)}$). The first entry of v, v_k, is normalized to a value of 1. In (2.11-2.13) a Householder reflection is applied to submatrix $S \in \Re^{(m-k)\times(n-k)}$, made up of entries $a_{k:m-1,k:n-1}$, so that the original submatrix S is substituted by the product $P \times S$ in order to zero entries $a_{k+1:m-1,k}$. This product is obtained by performing the operation $S + vw^T$ (2.13), where vector $w = \beta S^T v \in \Re^{n-k}$ (2.12) and the scalar $\beta = -2/v^T v$ (2.11). Norms are updated in (2.14). Thus, once all the iterations (n at most) of the algorithm have been carried out, the upper triangular matrix R is obtained, as shown in Figure 2.5.

$\mathrm{v}_k = 1.0;$
for (i=k+1; i<m; i++)
$$v_i = a_{i\,k} / (a_{k\,k} + sign(a_{k\,k}) \cdot \sqrt{norm_k}); \tag{2.10}$$

$$\beta = -2 \Big/ \sum_{i=k}^{m-1} v_i^2; \tag{2.11}$$

for (j=k; j<n; j++) {
$$w_j = \beta \cdot \sum_{i=k}^{m-1} a_{ij} \cdot v_i; \tag{2.12}$$
for (i=k; i<m; i++)
$$a_{ij} = a_{ij} + v_i \cdot w_j; \tag{2.13}$$
$$norm_j = norm_j - a_{k\,j}^2; \tag{2.14}$$
}

Figure 2.4: *Householder algorithm*

As in MGS, this is an $A \times \Pi = Q \times R$ factorization. Besides, the algorithm we have considered here is a column-oriented algorithm, but there are also row-oriented Householder transformations, which zero a designated subrow of matrix A. Another version of these algorithms is the *block Householder* QR factorization, in which the matrix to be decomposed is divided into blocks (this procedure is described in [42, Chapter 5]).

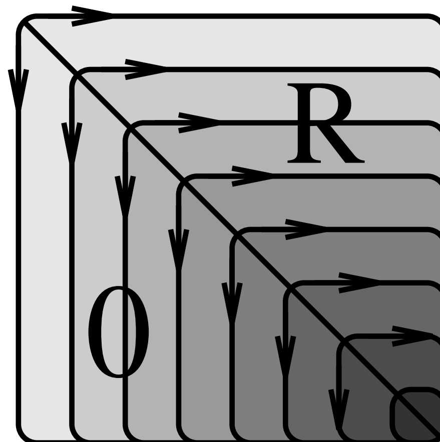

Figure 2.5: *Process of obtaining matrix R from the original matrix A*

In general, obtaining matrix Q in an explicit manner is not necessary. It can be obtained by previously storing it in a factorized format. This consists of storing the Householder vectors as they are obtained, in the lower triangular part of matrix R. And, from this factorized format, an algorithm (*backward accumu-*

for (i=m-1; i>k; i--)
Apply Givens rotation to subrows $a_{i-1,k:n-1}$ and $a_{i,k:n-1}$; (2.15)

for (j=k+1; j<n; j++)
$norm_j = norm_j - a_{k\,j}^2$; (2.16)

Givens rotation:

$$\begin{pmatrix} a'_{\alpha\,k} & a'_{\alpha\,k+1} & \cdots & a'_{\alpha\,n-1} \\ 0 & a'_{\beta\,k+1} & \cdots & a'_{\beta\,n-1} \end{pmatrix} \leftarrow \begin{pmatrix} gcos & -gsin \\ gsin & gcos \end{pmatrix} \cdot \begin{pmatrix} a_{\alpha\,k} & a_{\alpha\,k+1} & \cdots & a_{\alpha\,n-1} \\ a_{\beta\,k} & a_{\beta\,k+1} & \cdots & a_{\beta\,n-1} \end{pmatrix}$$

where $gcos = \frac{a_{\alpha\,k}}{\sqrt{a_{\alpha\,k}^2 + a_{\beta\,k}^2}}, \quad gsin = \frac{-a_{\beta\,k}}{\sqrt{a_{\alpha\,k}^2 + a_{\beta\,k}^2}}$ (2.17)

Figure 2.6: *Pseudo-code for Givens rotations*

lation) can be applied to get Q in an explicit way (see [42, Chapter 5] for more details).

2.4 Givens Rotations

Using Givens plane rotations, matrix Q is not explicitly calculated and matrix A is overwritten by matrix R (the same as in Householder). As we will show in Section 3.5, it is not necessary to calculate matrix Q to solve least squares problems by means of the QR decomposition.

Givens algorithm with column pivoting is built by substituting expressions (2.5-2.9) of the MGS algorithm (Figure 2.1) by the pseudo-code expressions (2.15) and (2.16) of Figure 2.6. As in Householder, note that only matrix A and column norms are involved in the pivoting (2.4).

Givens rotations are applied in (2.15) to zero the subdiagonal elements of column k. A Givens rotation involving two rows named α, β of matrix A consists of performing the update described in expression (2.17). The rotation is clearly orthogonal and it is not necessary to perform inverse trigonometric functions. As we can see, Givens rotations allow us to annihilate entries of matrix A more selectively and in a more flexible order than the other two QR algorithms. Finally, the norms of the updated columns are calculated in (2.16). Due to column pivoting, it is an $A \times \Pi = Q \times R$ decomposition.

The Givens algorithm considered here is the Classical Givens; there is another version named Givens-Gentleman (also known as fast Givens) [42, Chapter 5].

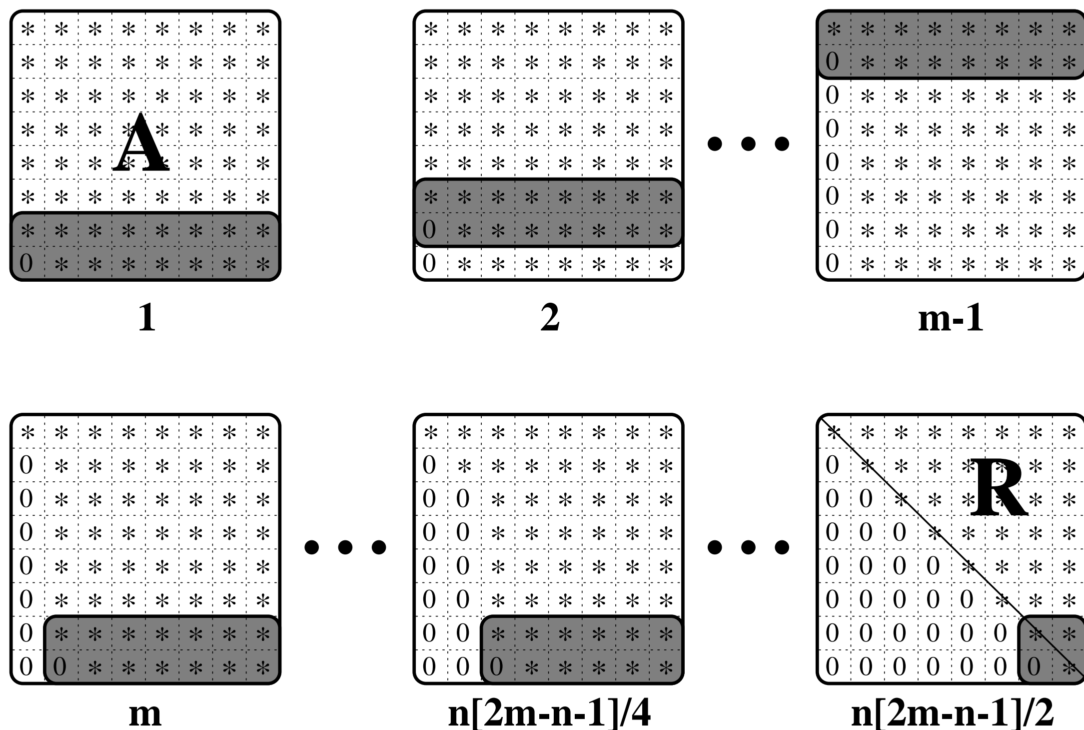

Figure 2.7: *Sequential Givens rotations*

The latter is more economic with regard to floating point operations, but it requires some kind of pivoting and scaling operations in order to prevent from a possible overflow. This overhead can degrade performance.

The Classical Givens algorithm upper triangularizes matrix A by introducing zeros column by column, as shown in Figure 2.7. The numbers in this figure indicate the number of rotations which have been necessary to achieve the corresponding stage shown by each picture.

There is another version which introduces zeros in A row by row [42, Chapter 5]. In this case, if we implement a pivoting mechanism, it should be a row pivoting.

2.5 QR Applications: Linear Least Squares Problems

The main use of QR decomposition algorithms lies in the various applications they have in linear algebra to solve linear systems of equations, least squares problems, eigenvalue and eigenvector problems, coordinate transformations, projections, linear programming, optimization problems, etc. It is necessary to solve these problems in many scientific areas, such as fluid dynamics, structural analysis, circuit simulation, device simulation, quantum chemistry $\cdots$ Therefore, high-quality software is necessary for those scientific numerical computations; for

example, a sparse QR factorization implementation in *Matlab* is described by Matstoms in [67].

The bibliography for QR applications is very extensive. Kaufman [60] presents an efficient parallel QR algorithm for determining all the eigenvalues of a symmetric tridiagonal matrix; Haag and Watkins [49] describe hybrid codes that combine QR and LR [42, Chapter 7] algorithms to calculate the eigenvalues and, optionally, eigenvectors of real, nonsymmetric matrices; Watkins [104] discusses the numerical stability of the QR factorization applied to the calculation of eigenvalues; Chandrasekaran and Ipsen [20] present an algorithm for computing the singular value decomposition of a real upper triangular matrix R that is based on the repeated QR decomposition of R; the work of Schreiber et al. [87] describe an efficient parallel QR-based algorithm to obtain all eigenvalues and all eigenvectors of a matrix, which can be applied to quantum mechanical calculations.

Regarding linear systems problems, there are many QR-based approaches; we just cite two for sparse matrices: [74][81]. Interesting references for sparse least squares problems are the papers of Matstoms [66][67]. There are also constrained and weighted least squares problems; in [48], an error analysis of this kind of problems is performed. Bendtsen et al. [12] describe, as an alternative to the classical simplex method, a new algorithm for linear programming based on a QR factorization.

As an application example of the QR decomposition, we approach a standard problem in linear algebra: the least squares problem. It consists in calculating a vector $x \in \Re^n$ that minimizes $\|Ax - b\|_2$ (euclidean norm), where $A \in \Re^{m \times n}$ $(m \geq n)$ and $b \in \Re^m$. If the rank of A is maximum (n), the least squares problem has one unique solution (x_{ls}). Otherwise, it has an infinite number of solutions x_{set}, one of them with minimum norm and which we will also denote as x_{ls}, $x_{ls} = x \in x_{set}$ *such that* $\|x\|_2$ *is minimum*. If m=n, the least squares problem is equivalent to solving a linear system of equations $Ax = b$, as $\|Ax - b\|_2 = 0$ (minimum norm). Sparse linear least squares problems appear in several technical and scientific areas such as geodesic survey, structural analysis, tomography, photogrametry, molecular structure $\cdots$

Several methods have been developed to solve this problem [41]: the augmentation method, the Peters-Wilkinson method, the method of normal equations and iterative methods. This problem can also be solved by adapting the algorithm that carries out the QR decomposition of a matrix A. In particular, the least squares problem is equivalent to solving the upper triangular system $R\Pi^T x = Q^T b$, by means of a back-substitution and a permutation of the entries of the solution vector x (due to the column pivoting) . This approach is adequate due to the good numerical stability of the QR factorization. If $rank(A) = n$, this algorithm calculates the one unique solution to the least squares problem. If $rank(A) < n$, only one of the infinite solutions is obtained, the one called basic solution, which has a maximum of *rank* nonzero elements and that, in general, does not coincide with the minimum norm solution x_{ls}.

2.6 A Sparse Approach: Data Storage Structures

A matrix is sparse if many of its elements are zero. The percentage of null elements a matrix must present in order to be considered sparse depends on the model or pattern of the nonzero elements, the algorithm to be carried out over the matrix, and even on the computer architecture. In general, we say that a matrix is sparse if it is advantageous to exploit its null elements with the development of a sparse version of an algorithm, instead of a dense one. Although a sparse QR factorization could be treated with a QR dense program, the cost of ignoring sparsity, both in storage and execution time, would erase the benefits of a sparse approach.

The choice of the storage scheme to support the sparse matrix is very important. Many different ways of storing sparse matrices have been developed to take advantage of the matrix pattern or the characteristics of the problems from which they arise [10][70][85]: Compressed Row/Column Storage (CRS/CCS), Block Compressed Row Storage, Compressed Diagonal Storage, Coordinate Format, Jagged Diagonal Format, Skyline Storage, Quadtree Representation [105], etc.

The main factor we have taken into account in the selection of the data storage structure has been the nature of the QR algorithms. As in the sparse QR factorizations the pattern of the matrix varies due to the fill-in (the appearance of new nonzero entries), it is convenient to use dynamic data structures (lists, trees $\cdots$) for the storage of the nonzero entries that support the changes in the pattern of the non-null elements. Thus, if we analyze the data flow of the algorithms, we only need column access to the matrix (except for Givens); this fact leads to a column-oriented data structure.

The CCS format is a column-oriented structure. It represents a sparse matrix A as a set of three vectors ($DATA$, ROW and COL); $DATA$ stores the nonzero entries of A, as if it were traversed in a column-wise fashion, ROW stores the row indices of the entries in $DATA$, and COL stores the locations in $DATA$ that start a column. By convention, we store in position n (in a C array) of COL the number of nonzero entries of A plus one. Due to the characteristics of our algorithms this data structure could be suitable for storing the sparse matrix A, but it is very costly to support the fill-in of the factorization with this structure: we would have to re-copy (using a buffer) the corresponding updated submatrix in each iteration k of our algorithms, which eliminates all the advantages in computation times of the sparse approach. Besides, we have to estimate the amount of memory necessary to perform the factorization, because these three vectors must be declared at compile-time.

To solve this drawbacks, we have chosen two data structures that are basically variants of the CCS format, and which are based on one-dimensional linked lists and packed (or compressed) vectors, respectively, as shown in Figure 2.8, together with their corresponding C declarations.

The lists could be singly-linked or doubly-linked. The use of doubly-linked lists accelerates the management of the data structure when carrying out operations

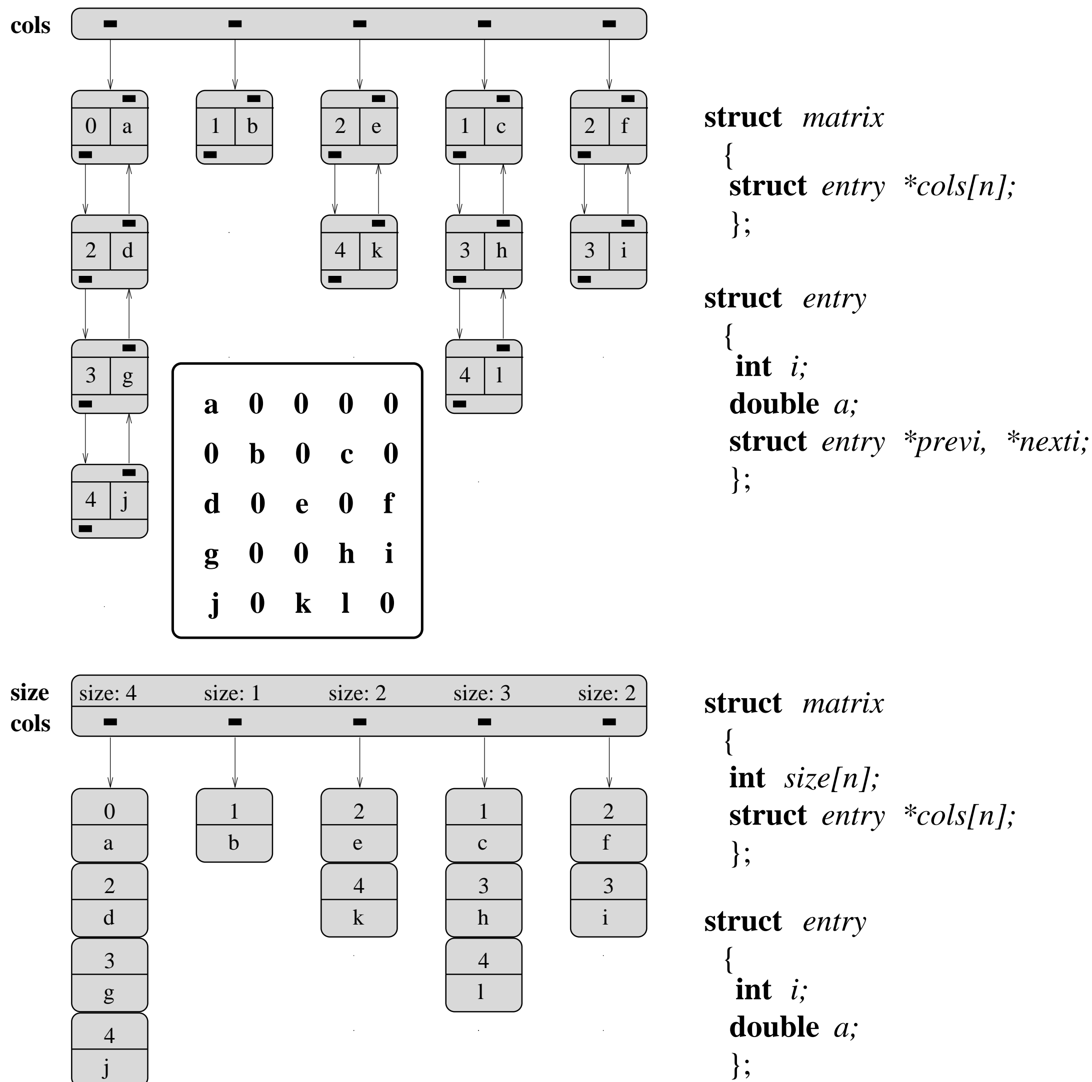

Figure 2.8: *Data structures for sparse matrices: doubly-linked lists and packed vectors*

such as insertions and deletions, but at the cost of more memory usage; besides, it allows list traversing in both directions. As there is currently a large amount of memory available in distributed memory parallel computers, we assume this larger memory usage in order to decrease the computational times. Consequently, we have chosen to use doubly-linked lists, each one of them represents one column of the matrix. As we can see in Figure 2.8, each item of the list stores the row index i, the matrix element a, and two pointers *previ, nexti* to the previous and to the next entry of the list, respectively. Lists are arranged in growing order of the row index; *cols* is an array of pointers to the beginning of each column (it has the same function as vector *COL* of the CCS format). Although it is not shown in this figure, there are also, for the doubly-linked list data structure, pointers to the end of each column, in order to allow traversing the columns of the sparse

matrix in reverse order. Obviously, this array is not necessary for the packed vector structure because the end of each packed vector can be reached by means of a sum operation of the corresponding entries of arrays *cols* and *size*.

The fact that we only require efficient access by columns (except for the Givens algorithm, as will be discussed in Section 3.4) implies large memory savings. Thus, a sparse LU decomposition [8][90] accesses data both by rows and columns, and row and column pivotings are performed; it requires a more suitable data structure, such as two-dimensional linked lists (see Figure 6.4). This structure stores, in addition to what we have indicated for one-dimensional lists, the column index of each element and two additional pointers (one to the previous row element and another to the next one). On the other hand, the time needed for managing this structure would increase. For instance, Stappen et al. [90] use two-dimensional linked lists, both ordered and unordered.

Our data structure based on linked lists has also severe drawbacks. The dynamic memory allocation/deallocation for each new entry is time consuming, causes memory fragmentation and spatial data locality loss, which makes traversing lists expensive due to cache misses.

Packed or compressed vectors eliminate the disadvantages of the lists. This structure is similar to the linked list one, but without the pointers *previ* and *nexti*. Therefore, this structure requires only around half as much memory space as the doubly-linked list, considering the fact that the *int* and *double* data types in C take up 8 bytes both (this is true for the C compilers we have used for our programs) and disregarding the space occupied by vector *size*. In addition to the array of pointers *cols*, we require the array named *size* which contains the number of nonzero elements of each column.

Using packed vectors, the elements of the columns are stored in adjacent memory positions. This exploits the locality, increases the cache hit ratio, reduces memory fragmentation and avoids having to go through the links of the lists to gain access to the elements; note that, using packed vectors, memory allocations and releases are performed once per every processed column, whereas by using linked lists these operations are performed once per every inserted or erased entry, respectively. Consequently, execution times are reduced, as we will test in Section 4.2.

We have programmed versions of the three algorithms using both data structures, except for Givens, which is only implemented with lists. It is important to remark that the linked list structure is more flexible and general than the packed vector structure, which can be used specifically for the MGS and Householder algorithms due to their characteristics but, in general, it is not always possible to use packed vectors for sparse factorizations involving fill-in. For instance, it is not possible to apply the packed vector approach described above in our Givens algorithm because there is a column swap, but fill-in is generated row by row (column by column in MGS and Householder), which makes a data structure (linked lists) that allows the insertion of an entry in the middle of each column necessary. We will return to this subject in Section 3.4.

2.7 A Strategy to Preserve Sparsity

Fill-in is a serious problem in our sparse QR algorithms because it can greatly reduce the advantages of the sparse approach in relation to the dense one. A high fill-in is not a desirable situation due to the increase in storage cost and computation time.

The factors that influence the fill-in of a sparse matrix are the following: the dimensions and rank of the matrix, the sparsity rate (percentage of null elements) and a factor of great importance but difficult to model, the pattern of the matrix, that is, the location of the nonzero entries. Thus, fill-in may vary significantly between two matrices with the same dimensions, rank and sparsity rate, depending on how nonzero elements are located. Given the pattern of the matrix A to be decomposed by means of the MGS procedure, the structure of the resulting matrices Q and R is determined in [51] by Hare et al.

There are reordering methods, as a first stage before the factorization, to avoid the increase in the sparsity rate of the matrix. For instance, Ostromsky et al. [75] have designed a parallel sparse Givens-based QR factorization algorithm, where matrix reorderings are applied in such a way that most zero entries are located in the lower triangular part of the matrix. Raghavan [81], on the other hand, utilizes a variant of the CND (Cartesian Nested Dissection) algorithm as a reordering heuristic with the aim of reducing fill-in in its row-oriented Householder-based sparse parallel algorithm. A row ordering strategy for Givens rotations, based on pairing rows to minimize fill-in, is described in [83]. There are similar reordering strategies for other factorizations (for instance, [62] for Cholesky factorization). Many of these strategies based on reorderings are sequential by nature (that is, it is not possible to parallelize them), or are specific for a concrete family of sparse matrices, or are highly algorithm-dependent.

Usually, the reordering strategies for sparse matrices imply the use of graphs and are closely tied to multifrontal techniques [33, Chapter 10], which basically consist in a statement of sparse numeric computations as a sequence of dense matrix operations (that is, small dense submatrices are exploited) and can be applied to many problems (LU and Cholesky factorizations, for instance). This approach was later used for orthogonal factorizations (Householder and Givens): [65][66][81]. As an example, the second reference consists in a multifrontal Householder procedure which uses an elimination tree to efficiently describe the structural relationships between the columns of the coefficient matrix A. The multifrontal methods for the QR factorization are not considered in this work.

2.7.1 A New Column Pivoting Criterion

It would be of interest to implement a simple method to preserve sparsity during factorization in the sparse QR algorithms. The most common heuristic strategy used for maintaining the sparsity rate in the LU factorization is the Markowitz criterion [33, Chapter 7]. In addition, numerical stability must be ensured in the

LU factorization by avoiding the selection of those pivots with a low absolute value. Since QR decomposition is more expensive than LU factorization, it must be carefully implemented to be competitive as a general solution technique.

We propose a general heuristic to preserve sparsity in the QR decomposition by taking advantage of column pivoting. This is, in contrast to reordering methods and symbolic preprocessings, a dynamic strategy, which is applied as the factorization is carried out. Besides, it is applicable to our three QR algorithms and to any kind of large sparse matrices (which often arise in scientific applications): symmetric or unsymmetric; random, band or block-structured, etc. Let us analyze our algorithms to justify this criterion.

In MGS each element of row k of matrix R (r_{kj}) is, basically, a dot product between the normalized pivot column (which becomes column k due to the swapping) and column j of matrix A (see Figure 2.1, expression (2.7)). If the pivot column has few nonzero elements, the probability of r_{kj} being 0 raises and, consequently, fill-in in R is reduced, as well as in matrix Q, because row k of matrix R is used to update matrix Q (expression (2.9)).

Therefore, in order to decrease fill-in, we propose a new criterion to select the pivot column, instead of the pivoting strategy commonly used in dense factorizations and described in expression (2.2). The new pivoting strategy is expressed as follows:

Find p, $k \leq p < n$, such that

$$\left\{ \epsilon \left(\frac{zero_p}{\max\limits_{k \leq j < n} zero_j} \right) + (1-\epsilon) \left(\frac{norm_p}{\max\limits_{k \leq j < n} norm_j} \right) \right\} \text{ is maximum and } norm_p \geq \phi \quad (2.18)$$

where $zero_j$ is the number of zero elements in column j ($a_{0:m-1,j}$), $zero_p$ is the number of zero entries in column p ($a_{0:m-1,p}$)and ϵ, $0 \leq \epsilon \leq 1$, is a prefixed fill-in parameter.

The first term of the sum refers to fill-in reduction, whereas the second one relates to numerical stability. Obviously, for $\epsilon = 0$, the pivot column selection criterion is equivalent to the one described in (2.2) for dense matrices. Although we try to reduce fill-in as much as possible by choosing a pivot column with few nonzero elements (by setting $\epsilon \approx 1$), we shall always maintain a reasonable degree of numerical stability in the algorithm by discarding as pivot columns those with the square of the norm close to zero (that is, $norm_p < \phi$).

With regard to Householder reflections, we should select a pivot column p with many zero entries in subcolumn $a_{k:m-1,p}$ (subdiagonal entries), since the Householder vector v, which is used to upper triangularize matrix A (see Figure 2.4, expression (2.13)), is obtained by dividing that subcolumn by a certain value (2.10). This way, v is a very sparse vector, and the probability of entries of vector w being 0 increases, as w is calculated using vector v (2.12); and, finally, if vectors v and w have many zero entries, as matrix A is updated using both vectors (expression (2.13)), fill-in may be reduced in this matrix.

Matrix	Origin	m×n	#A	% #A
SHL400	Linear programming problems	663×663	1712	0.39%
JPWH991	Circuit physics modelling	991×991	6027	0.61%
SHERMAN1	Oil reservoir modelling	1000×1000	3750	0.37%
MAHINDAS	Economic modelling	1258×1258	7682	0.49%
ORANI678	Economic modelling	2529×2529	90158	1.41%
SHERMAN5	Oil reservoir modelling	3312×3312	20793	0.19%

Table 2.1: *Harwell-Boeing sparse matrices*

We should also choose, in Givens rotations, a pivot column p with many null elements in subcolumn $a_{k:m-1,p}$ because it is not necessary to rotate the rows corresponding to the zero elements of this subcolumn (see Figure 2.7). That is, the purpose of one rotation is to annihilate one entry of the matrix and, if this entry is zero, it is not necessary to apply this rotation; this fact saves a lot of computations and avoids the fill-in which could appear in the entries $a_{\alpha,k+1:n-1}$, $a_{\beta,k+1:n-1}$ if the rotation were applied (see Figure 2.6, expression (2.17)).

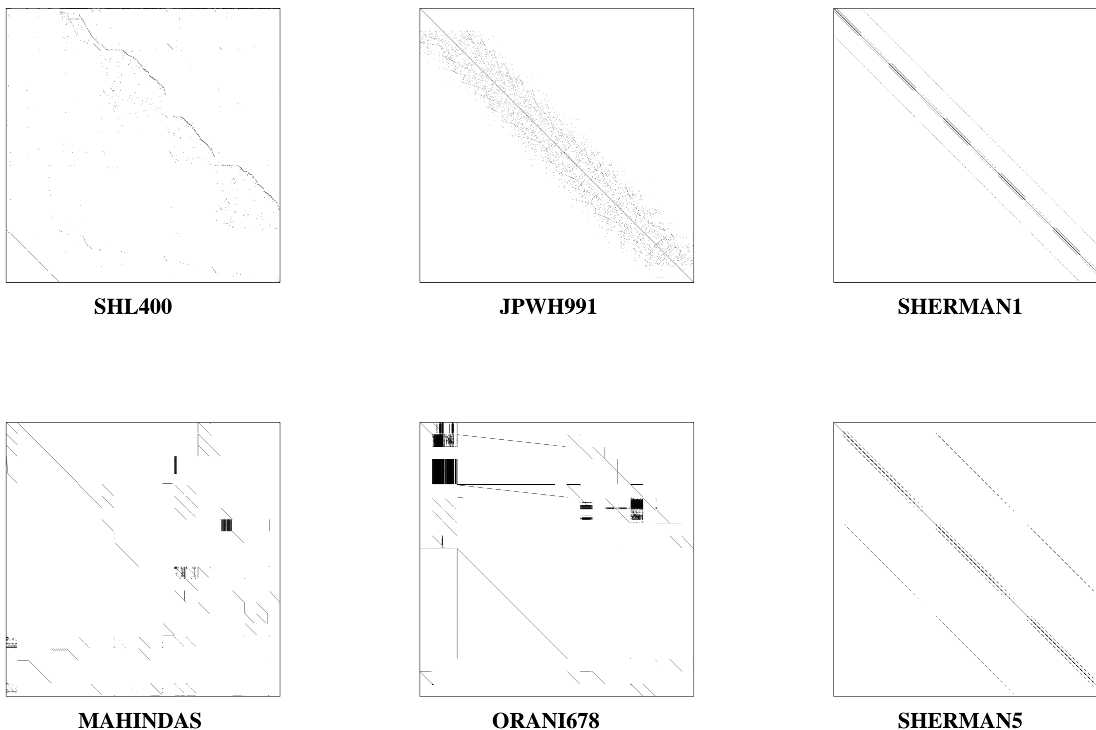

Figure 2.9: *Patterns of the Harwell-Boeing test matrices*

As a conclusion, the strategy to preserve sparsity described in (2.18) can be applied indistinctly to the three QR algorithms, but for Householder and Givens factorizations the term $zero_j$ refers to the number of zero elements in subcolumn j, defined as $a_{k:m-1,j}$; and $zero_p$ is the number of zero entries in subcolumn

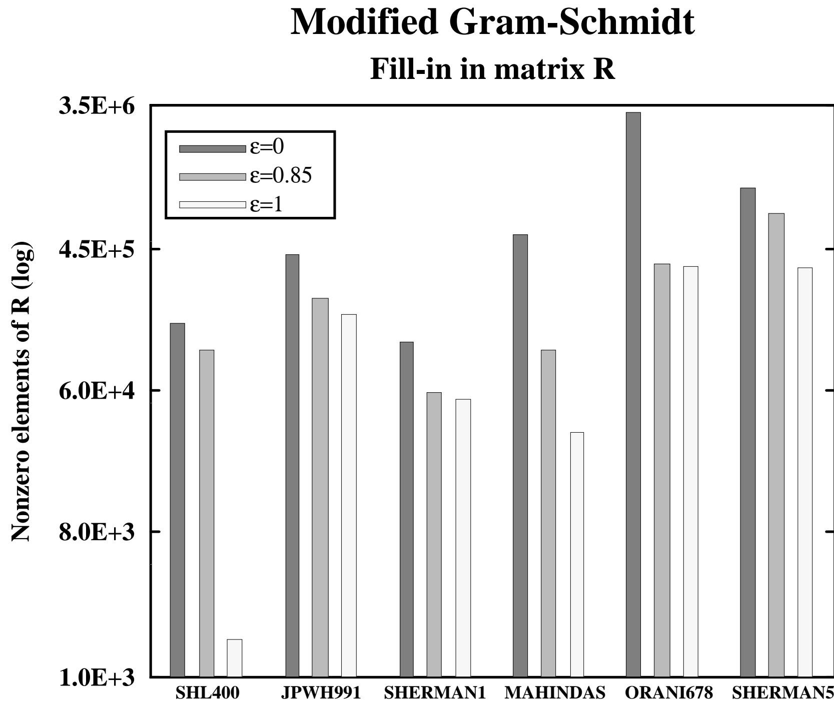

Figure 2.10: *MGS: fill-in in matrix R for different values of* ϵ

$a_{k:m-1,p}$. In [94], we focus on this strategy.

With the aim of testing this criterion, we have run the sparse sequential algorithms on a SPARC workstation, using six matrices from the Harwell-Boeing sparse matrix collection [34], whose patterns are shown in Figure 2.9. These matrices have different patterns, in order to show that the criterion (2.18) fits well any kind of sparse matrix. A description of these matrices is presented in Table 2.1, where *Origin* indicates the scientific discipline in which the corresponding matrix was obtained; $m \times n$ are the dimensions of the matrix, $\#A$ is the number of nonzero elements of matrix A and $\%\#A$ is the percentage of these elements, that is, the sparsity rate.

Figures 2.10, 2.11 and 2.12 show fill-in in matrix R after the factorization (note that the scale of these graphs is logarithmic), for MGS, Householder and Givens, respectively, with $\epsilon = 0$, $\epsilon = 0.85$ and $\epsilon = 1$ in expression (2.18). As can be observed, if we follow the common strategy of pivoting in dense matrices ($\epsilon = 0$), fill-in is very high. This situation eliminates all the advantages of a sparse approach because storage costs and computation times are high. As ϵ increases, fill-in is greatly reduced, and we obtain acceptable results for $\epsilon = 1$. As we can see, by making $\epsilon = 1$, fill-in decreases with respect to $\epsilon = 0$ by 77% (MGS), 65% (Householder) and 78% (Givens) on average for this set of sparse matrices, which is a very noteworthy reduction. The percentage of reduction, matrix by matrix is: *SHL400* (99% for MGS, 99% for Householder and 99% for Givens), *JPWH991* (59%, 40% and 59%), *SHERMAN1* (55%, 24% and 56%), *MAHINDAS* (94%, 90% and 94%), *ORANI678* (89%, 83% and 89%), *SHERMAN5* (68%, 55% and

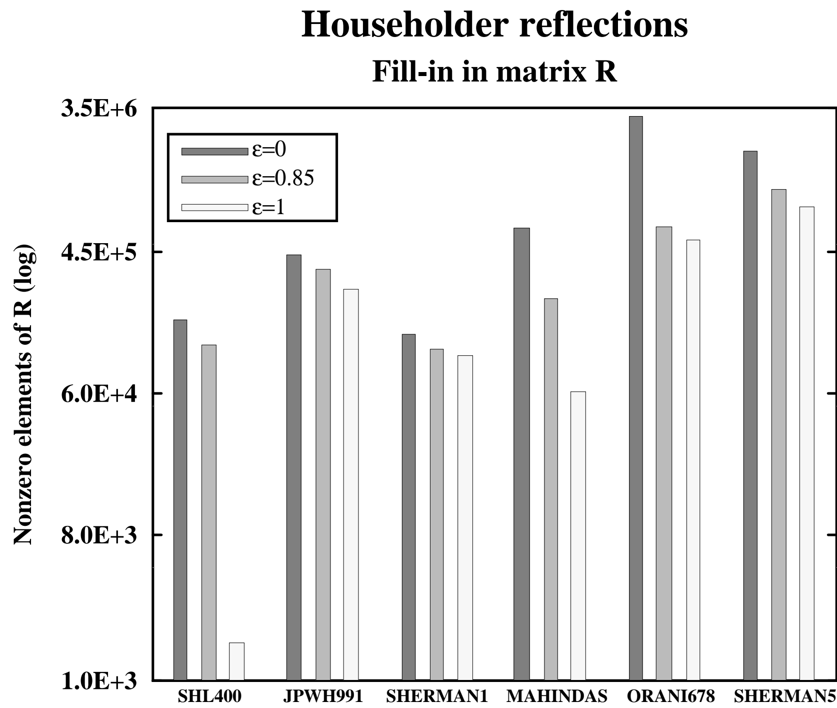

Figure 2.11: *Householder: fill-in in matrix R for different values of* ϵ

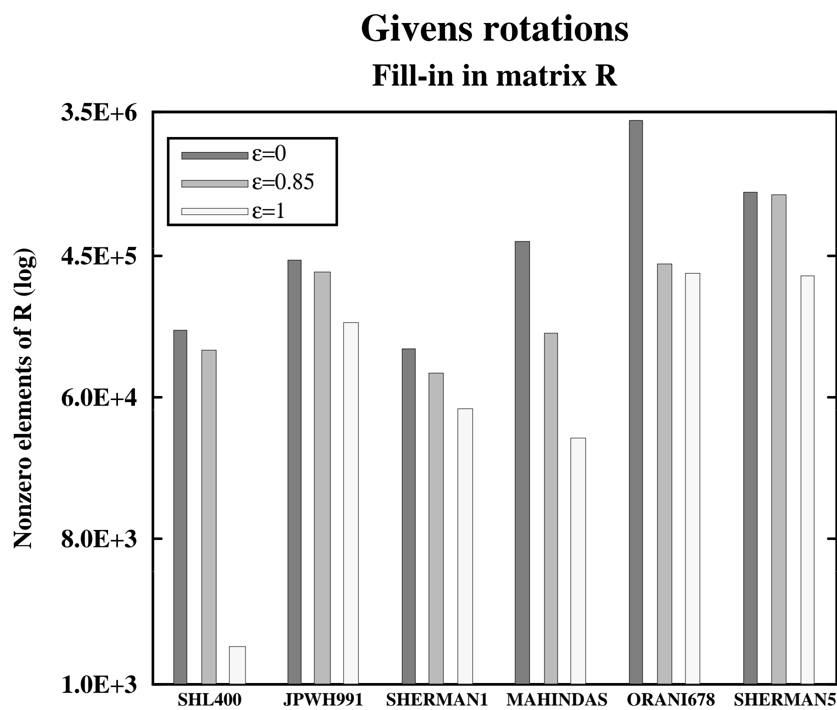

Figure 2.12: *Givens: fill-in in matrix R for different values of* ϵ

69%, respectively).

In general, we can assert that MGS and Givens are the algorithms which produce the least fill-in (both algorithms remain at very similar levels), whereas Householder produces the highest fill-in.

As the fill-in of the coefficient matrix increases during the factorization, there are techniques based on switching to a dense factorization at a certain point. Hence, a sparse factorization code is initially executed, but when the switch point is reached, a dense code continues the factorization (the corresponding submatrix is scattered to a dense array). This strategy is described in [33, Chapter 9] for an LU factorization.

2.7.2 Numerical Analysis

Next, we will discuss the effect of this reduction of fill-in on the accuracy of the results obtained in solving the least squares problem by means of QR factorization. As we described in Section 2.5, if the coefficient matrix A is square, the problem of solving least squares is equivalent to solving the linear equation system $Ax=b$. If the rank of A is maximum, then there is only one unique solution. To measure the accuracy of the results, we used the matrices of Table 2.1 (which are maximum-rank square matrices), as well as a vector b chosen so that the exact solution of the system is a vector x whose components are all one ($x = \vec{1}$). Let us define the error as $\max_{0 \leq j < n} |1.0 - x_{sol}|$, where x_{sol} is the solution of the system, obtained by means of QR factorization. We used double precision floating point numbers, and those elements with an absolute value of less than $\phi = 10^{-20}$ were considered null. Raghavan [81] implemented programs for orthogonal factorization in single precision. Nevertheless, the use of numbers in single precision draws, for our algorithms, very inaccurate results and can also cause the rank of the matrix to be determined incorrectly. Ostromsky et al. [75] use, for their Givens algorithm, a threshold $\phi = 2^{-5}$ and the solution to the least squares problem is refined by means of an iterative method, such as the Conjugate Gradient method [10].

In MGS and in Householder the norm of the column being processed is used as a divider in Figure 2.1, expression (2.6) and in Figure 2.4, expression (2.10), respectively. This norm is calculated in each iteration as noted in (2.8) and (2.14), and may lead to a loss of accuracy. For this reason, in these two algorithms, we recalculate in each iteration the norm of the column k being processed with the aim of obtaining greater accuracy in the results. The computational cost of this calculation operation is negligible.

As a comparison, in Figures 2.13-2.18 we present the numerical errors for the different algorithms going from $\epsilon = 0$ (dense matrix criterion) up to $\epsilon = 1$, with a step $\Delta = 0.05$. We note that with the fill-in reduction strategy ($\epsilon = 1$), not only do we maintain accuracy in the results, but also we notably improve them on some occasions. As an example, when $\epsilon = 1$, the error is exactly 0 for matrix *SHL400*, for all three methods; in the case of matrix *ORANI678*, when using

MGS, the error is lower by 3 orders of magnitude with respect to $\epsilon = 0$. We obtain the greatest accuracy, with fill-in reduction, for matrix *SHL400* and the least accuracy for matrix *MAHINDAS*.

On the whole, for this set of test matrices, the least accurate results were obtained using the MGS algorithm. Nevertheless, the errors are very similar for the Householder and Givens algorithms.

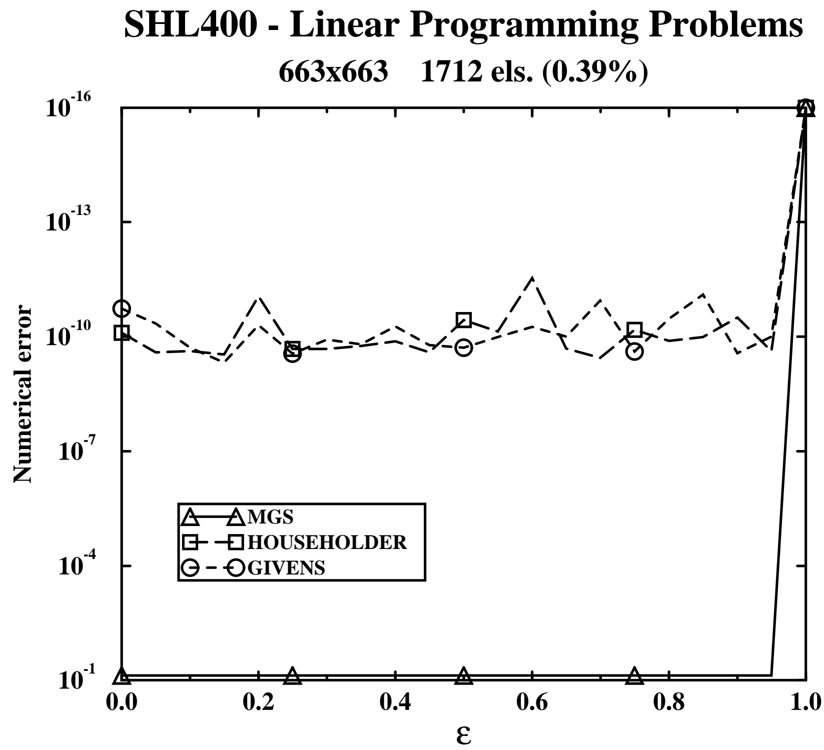

Figure 2.13: *SHL400: numerical errors for different values of* ϵ

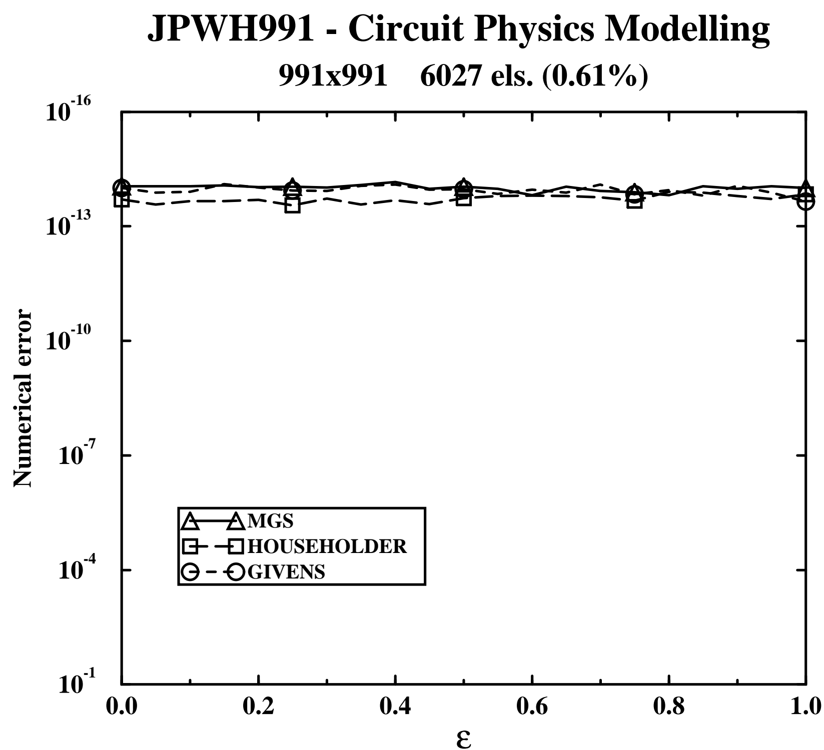

Figure 2.14: *JPWH991: numerical errors for different values of* ϵ

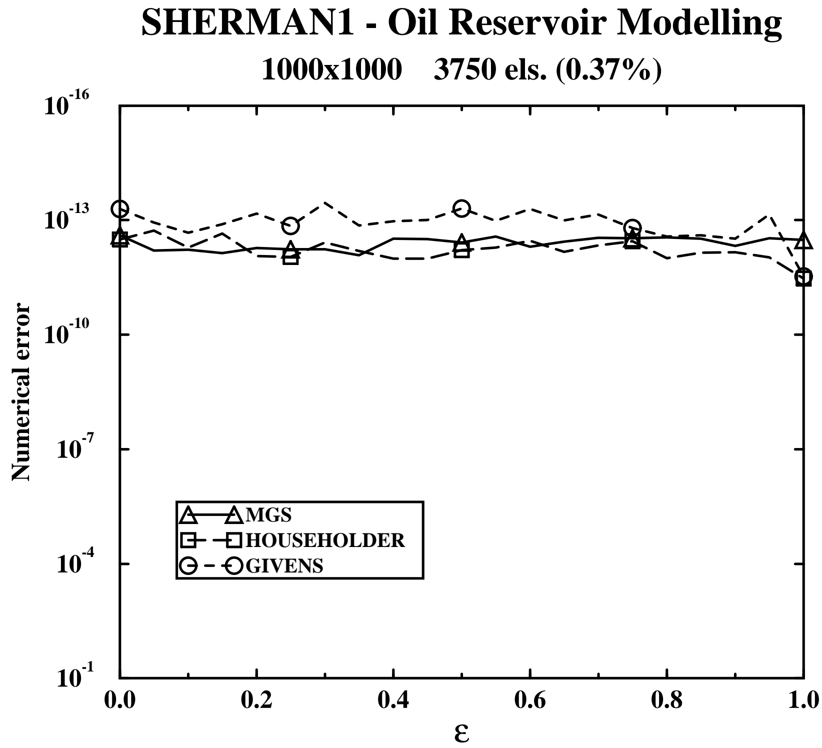

Figure 2.15: *SHERMAN1: numerical errors for different values of* ϵ

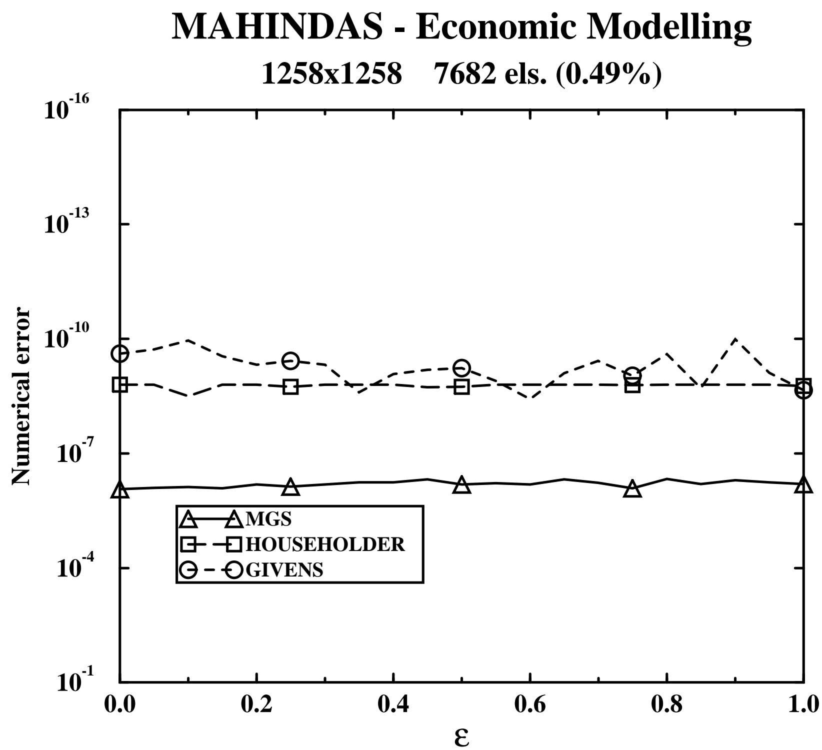

Figure 2.16: *MAHINDAS: numerical errors for different values of* ϵ

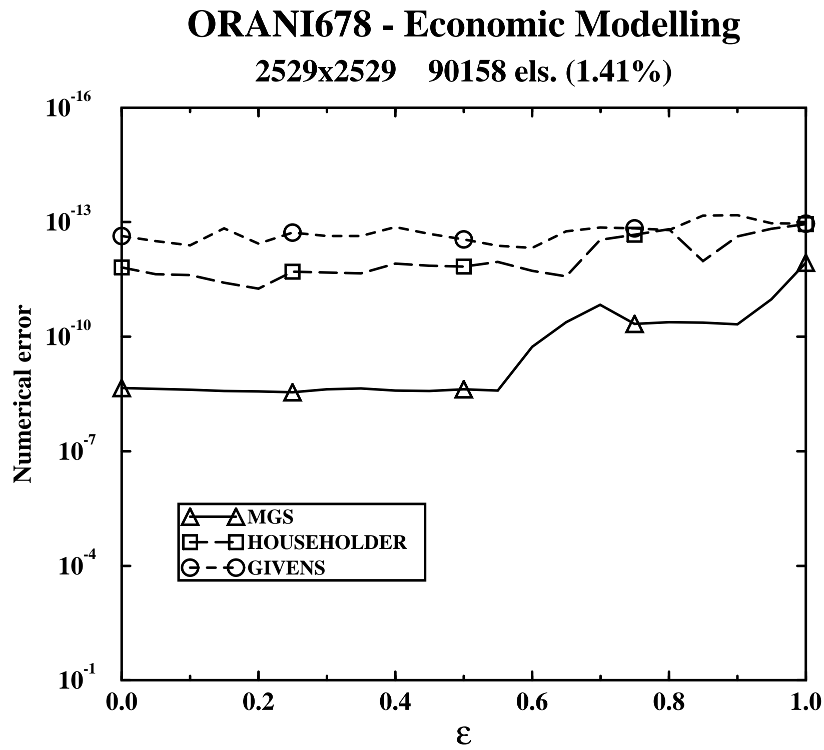

Figure 2.17: *ORANI678: numerical errors for different values of* ϵ

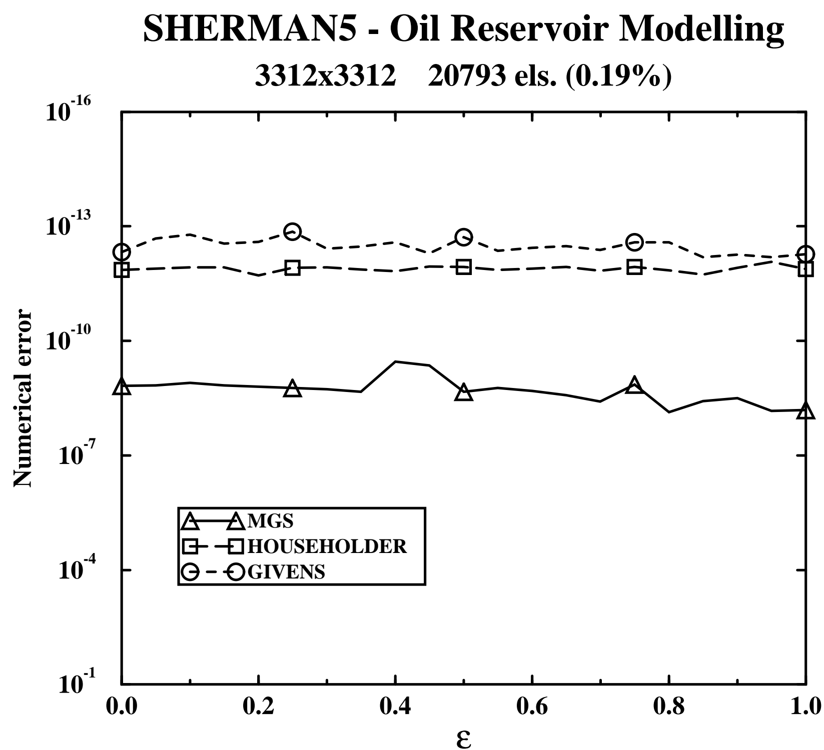

Figure 2.18: *SHERMAN5: numerical errors for different values of* ϵ

Chapter 3

Parallel QR Algorithms

3.1 Data Distribution

The selection of an appropriate data distribution is critical to achieve efficient parallel algorithms. When parallelizing algorithms with regular accesses to data (for instance, dense matrix codes), the most known data distributions are the block and cyclic ones, or even a combination of both (block-cyclic); for instance, the *block-cyclic(x)* distribution means that each Processing Element (PE) receives x contiguous elements of an array starting on PE #0; if $x = 1$, it is equivalent to the cyclic distribution. These dense data distributions are suitable for a wide range of dense matrix algorithms.

However, these distributions are not appropriate for our algorithms, with irregular patterns for accessing data. Our sparse codes contain array indirections (due to the data storage schemes) that produce not well-balanced parallel codes and complex communication patterns when using dense data distributions. An approach to deal with this kind of data accesses is the use of pseudo-regular data distributions as extensions of the classical block and cyclic regular distributions, such as MRD (Multiple Recursive Decomposition) and BRS/BCS (Block Row/Column Scatter). A complete explanation of these distributions, with their statistical properties can be found in [6][84]. The MRD scheme is, basically, a kind of block distribution for sparse matrices and is very adequate for molecular dynamics applications [100]. The BRS/BCS schemes are a generalization of the cyclic distribution for sparse matrices: the entries of the sparse matrix are distributed over the PEs in a cyclic way (per rows and per columns) onto a mesh of processors, as if the matrix were dense, but only the nonzero elements are stored using the CRS format (for BRS) or the CCS format (for BCS) (see BRS in Figure 6.3). Thus, the difference between BRS and BCS is that the former is row-oriented and the latter is column-oriented.

These distributions, which constitute a coordinate bisection scheme to decompose the sparse (irregular) data domain, are adequate for a great variety of sparse problems. There are more complex strategies to partition irregular domains for specific applications, such as graph bisection (used for finite element

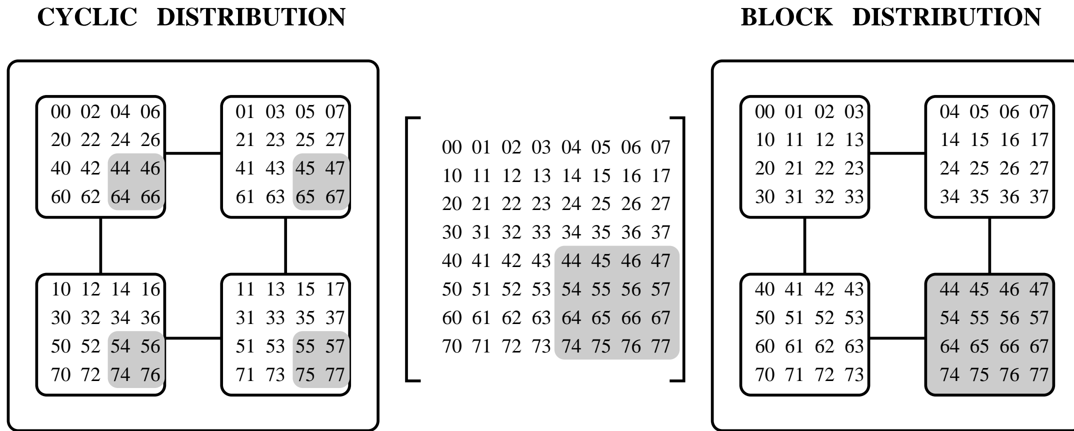

Figure 3.1: *Load balancing using a cyclic distribution for Householder reflections*

problems) and the spectral bisection (implemented in the CHAOS library [86]; CHAOS routines will be employed in some experiments of Chapter 7).

We have chosen a cyclic data distribution (also known as grid distribution and scattered square decomposition). The selection of this distribution responds to two reasons: data balancing and load balancing.

The first reason arises from the nature of the sparse matrices. As we work with general sparse matrices (we assume that they do not follow a particular model or pattern), the cyclic distribution is the most appropriate one in order to provide good data balancing. Let us assume that our sparse matrix contains areas or submatrices with large densities (for instance, matrix *ORANI678* of Figure 2.9). The use of a block distribution concentrates this large volume of data in few PEs and, thus, the unbalance in the data distribution is evident. We must point out that we work with general matrices, but if we worked with known families of sparse matrices, with a given location for the nonzero entries, the best strategy would be to preprocess the matrix (by means of row and column swaps) in order to obtain an adequate matrix pattern and, based on it, apply the most suitable distribution [33, Chapter 8].

The second reason is based on the characteristics of our algorithms, which are problems of reducing the index space. The three algorithms are made up of as many steps as columns in the coefficient matrix A (if it is maximum-rank). In iteration k of the main loop of the algorithms, entries $a_{0:m-1,k:n-1}$ (matrix Q) and $r_{k,k:n-1}$ (matrix R) are updated in MGS (see Figure 2.3); and entries $a_{k:m-1,k:n-1}$ (matrix R) are updated using Householder reflections or Givens rotations (see Figures 2.5 and 2.7). It is clear that if we employed a block data distribution, as the algorithms were executed, a large number of PEs (assuming a mesh topology) would be inactive because the entries of matrix A to be updated would be concentrated in a few PEs, due to the fact that the index space is progressively reduced through iterations. This is prevented using a cyclic distribution which, for our algorithms, provides a better load balance without any need for increasing communications. This fact is shown graphically in Figure 3.1, in which we have

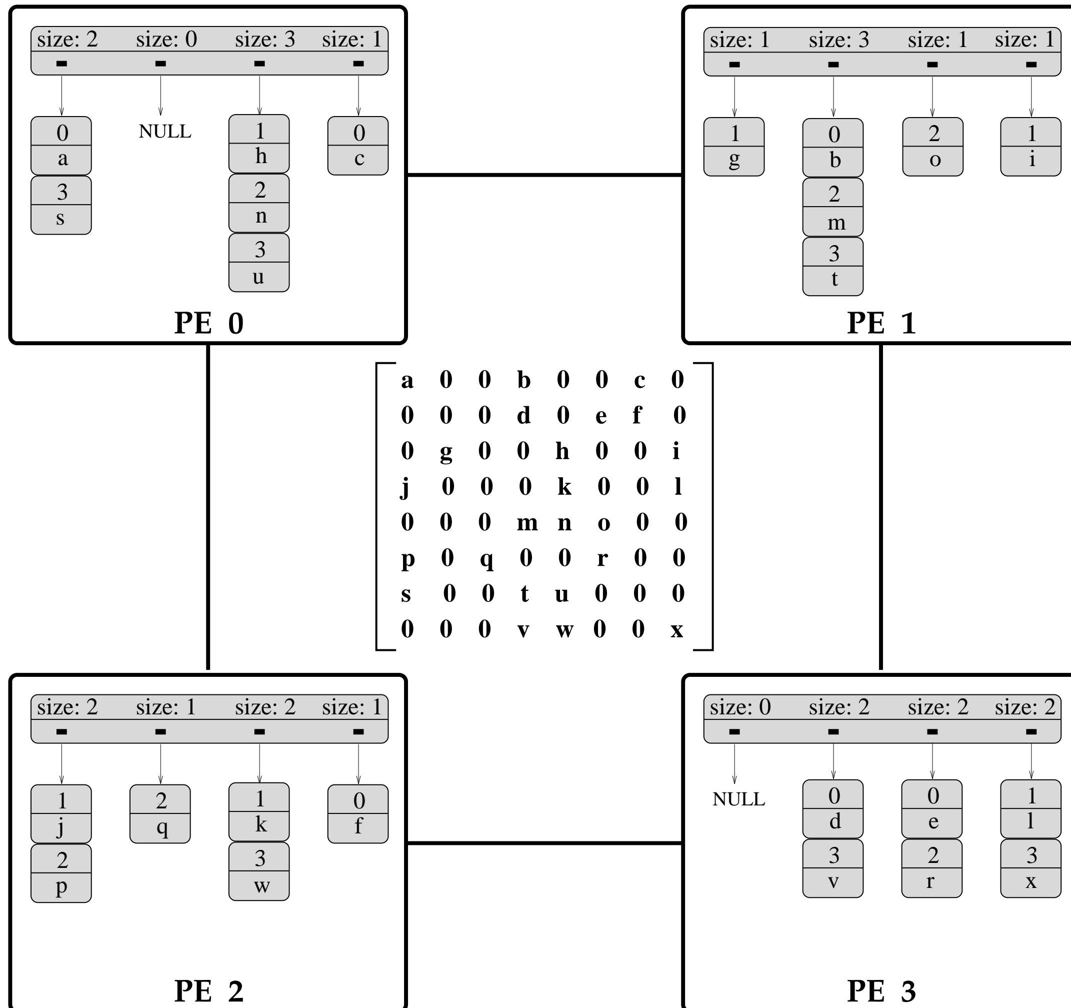

Figure 3.2: *Sparse matrix data distribution scheme*

a 2×2 mesh and an 8×8 matrix A; we assume that iteration $k = 4$ of the main loop of the Householder algorithm is being performed. The shaded part represents submatrix S (see Section 2.3), which is the portion of matrix A that is currently being processed.

Thus, as our matrices are sparse and the cyclic distribution is very suitable for our algorithms, we can use the BRS/BCS schemes. As we explained in Section 2.6 the algorithms only require efficient access by columns (except in Givens); therefore we select the BCS scheme. But the CCS format is not appropriate to support operations such as fill-in and column swapping and we use the data structures of Figure 2.8. As a conclusion, the data distribution used for the three algorithms is the same as a BCS scheme, but the data storage structure for the nonzero elements are linked lists or packed vectors instead of the CCS format (three vector structure).

Figure 3.2 shows the selected distribution for an 8×8 matrix onto a 2×2 mesh, with packed vectors as data structure (it would be the same scheme for linked lists, as the one shown in Figure 5.1). As we can see, nonzero entries of the

matrix are distributed over the PEs in a cyclic way (per rows and per columns) onto a mesh, as if the matrix were dense, but only the nonzero elements are stored using packed vectors. Once the algorithms have been executed, we obtain the resulting matrix R distributed in the same way.

We assume that, for all of our parallel algorithms, the coefficient matrix A is distributed onto a mesh with $npey \times npex$ PEs. Each PE is identified by coordinates $(pidx, pidy)$, being $0 \leq pidx < npex$ and $0 \leq pidy < npey$. By means of the data distribution, entry (I, J) of A is located in PE $(pidx, pidy) = (J\ mod\ npex,\ I\ mod\ npey)$. As the parallel algorithms described in the next sections are programmed for MIMD distributed memory computers (see Table 4.1), it is more appropriate to work with local indices for the matrices instead of global indices, as shown in Figure 3.2. Therefore, from the global indices (I, J) that identify the entries of the matrix, we can obtain the local indices (i, j) in each PE: $i = \lfloor \frac{I}{npey} \rfloor$, $j = \lfloor \frac{J}{npex} \rfloor$. In a similar way, we can reconstruct the global indices of the matrix from the local indices: $I = (i \times npey) + pidy$, $J = (j \times npex) + pidx$. This is necessary for the result collection stage.

3.2 Parallel MGS

The bibliography for parallel MGS algorithms is scarce. Zapata et al. [107] and Waring et al. [103] programmed MGS parallel algorithms on an nCUBE/10 (hypercube topology) and on a network of transputers configured as a simple pipeline topology, respectively, but for dense matrices.

Let us consider the sequential algorithm in Figure 2.1, but with a sparse approach, to see how it can be executed in parallel. First, each PE obtains the local norms (corresponding to the column segments it contains) in a vector named *norm*; by means of a reduction instruction (sum by columns, *y_sum*), the vector *norm* of each column of PEs will contain the norms of the corresponding global columns (2.1). As can be deduced, *norm* is a dense vector distributed over the X dimension of the mesh and replicated on the Y dimension, so that each row of PEs stores a copy of the complete vector. Then, the local maximum norm of each PE is obtained (this value is the same for each column of PEs). The global maximum (the pivot element) is obtained by means of a reduction routine which finds the maximum norm by rows of PEs (*x_max*). As a result, the pivot element, as well as p (the index of the pivot column) will be contained in all the processors (2.2).

The parallelization of the strategy to control fill-in (2.18) requires more communications than (2.2), due to reduction operations, but it is not very costly from a computational point of view. With this purpose, we have developed specific reduction operations, suitable for our pivoting criterion, in order to reduce latencies (start-ups), by means of grouping together the messages to be sent in these operations. For example, our codes have a reduction routine which obtains, at the same time, the minimum value of an integer vector and the maximum value of a floating point vector (in double precision), being these vectors cyclically distributed

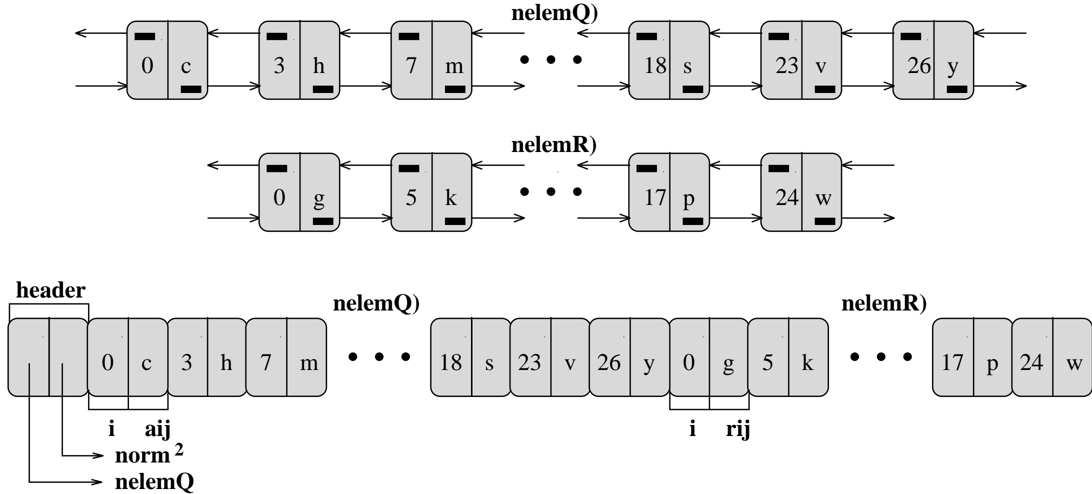

Figure 3.3: *Pivoting: buffer for sending local lists*

on each row of PEs.

In regard to column pivoting (2.4), we find three situations:

- If $k = p$, no action is taken.
- If $k \neq p$ and the corresponding columns are in the same PEs, then the column swap is local and is performed by means of an exchange of the corresponding pointers of array *cols* (see Figure 2.8).
- If $k \neq p$ and the corresponding columns are located in different columns of PEs, message-passing is required.

Regarding the third situation, the column of PEs that contains column k must exchange this column of matrices Q and R, as well as the corresponding norm, with the column of PEs that contains pivot column p. In order to send this information, we have to indicate the start memory address where this information is located and its size, that is, the information must occupy adjacent memory positions, a situation that does not happen with a list. To solve this drawback we use a packed vector that acts as a buffer for sending the required information. As we show in Figure 3.3, the information is stored in adjacent positions and the corresponding column of Q and R, as well as the square of the norm, are sent in a single message. If we established as many connections (for sending data) as entries of the column to be sent, it would imply a great temporal overhead. For instance, in the Fujitsu AP1000 computer (consult Section 4.1), the communication time T can be modelled by an affine function so that for a message of N bytes, $T = t_0 + t_b \times N\mu s$, being t_0 a fixed overhead in establishing the communication (start-up time or latency) and t_b the data transfer time per byte. In point to point communications (using the AP1000 *T-net*), $t_0 = 6.9\mu s$ and $t_b = 0.069\mu s$; in the *B-net*, these values vary depending on several conditions: $t_0 = 43 - 60\mu s$

and $t_b = 0.026 - 0.042\mu s$. In [58] we find numerous experiments and measures of the communication times for the AP1000. By sending all the information in a single message, the fixed overhead associated to each one of the communications is avoided.

Returning to the buffer of Figure 3.3, the procedure for moving the local list to the packed vector for the PEs which store column k or column p is the following: the local list corresponding to the column of matrix Q (and, later, R) is traversed from the beginning; as it is traversed, each one of the entries is stored in the buffer and erased from the list by means of a specific routine for deleting the first element of the list. As header of this exchange buffer, we indicate the value of the square of the norm of that column and the number of nonzero entries of matrix Q (*nelemQ*), in order to reconstruct the corresponding columns (local lists) of Q and R in the receiving PE by means of a routine for insertion at the end of the list (append). Obviously, with the parallel implementation which uses packed vectors as data structure for matrix A, we have the information ready to be sent, as it occupies adjacent memory positions and we do not need the buffer of Figure 3.3.

We will return to the swap procedure in Section 4.4, where a strategy of local pivoting is presented.

With regard to the parallelization of the MGS procedure (which we describe in detail in [28]), the current column k of matrix Q being processed (which was the pivot column before swapping) is normalized by dividing it by its norm (2.6). This normalized column is stored in a packed vector named *vcol*, in order to save computations in the subsequent operations. The column of PEs which contains the normalized column broadcasts it (stored in *vcol*) to all the corresponding PEs on the X axis (*x_brd*). This is shown in Figure 3.4a, for a 3×3 mesh and a 9×9 matrix. Therefore, *vcol* is a packed vector which is distributed over the Y dimension of the mesh and replicated on the X dimension, so that each column of PEs stores a copy of the complete vector. Then, for each column with a global index $\geq k$, we store in a vector named *vsum* the sparse local dot product of each column with the normalized current column previously stored in *vcol* (Figure 3.4b). Finally, the global dot product for each column is obtained in *vsum* by means of a reduction instruction (*y_sum*), as shown in Figure 3.4c; *vsum* is a dense vector with many zeros, which is distributed and replicated in the same way as vector *norm*. Observe that, in Figure 3.4, the arrays which have the same subscript have the same content (that is, they are replicated). The elements of *vsum* constitute the sparse row $r_{k,k:n-1}$ of matrix R (2.5), (2.7) (Figure 2.1). In the list implementation, row access to matrix R is not necessary to insert this new row, because it is an insertion made at the end of the corresponding lists (columns) with index j, being $vsum_j \neq 0$, $k \leq j < n$, which means column access.

The next step is updating the entries $a_{0:m-1,k:n-1}$ of matrix Q; so, a_{ij} is overwritten with $a_{ij} - vcol_i \times vsum_j$ (2.9). Note that it is a local operation and communications are not required. This operation is performed in parallel by all the PEs, by means of an outer loop which traverses each column (list) of

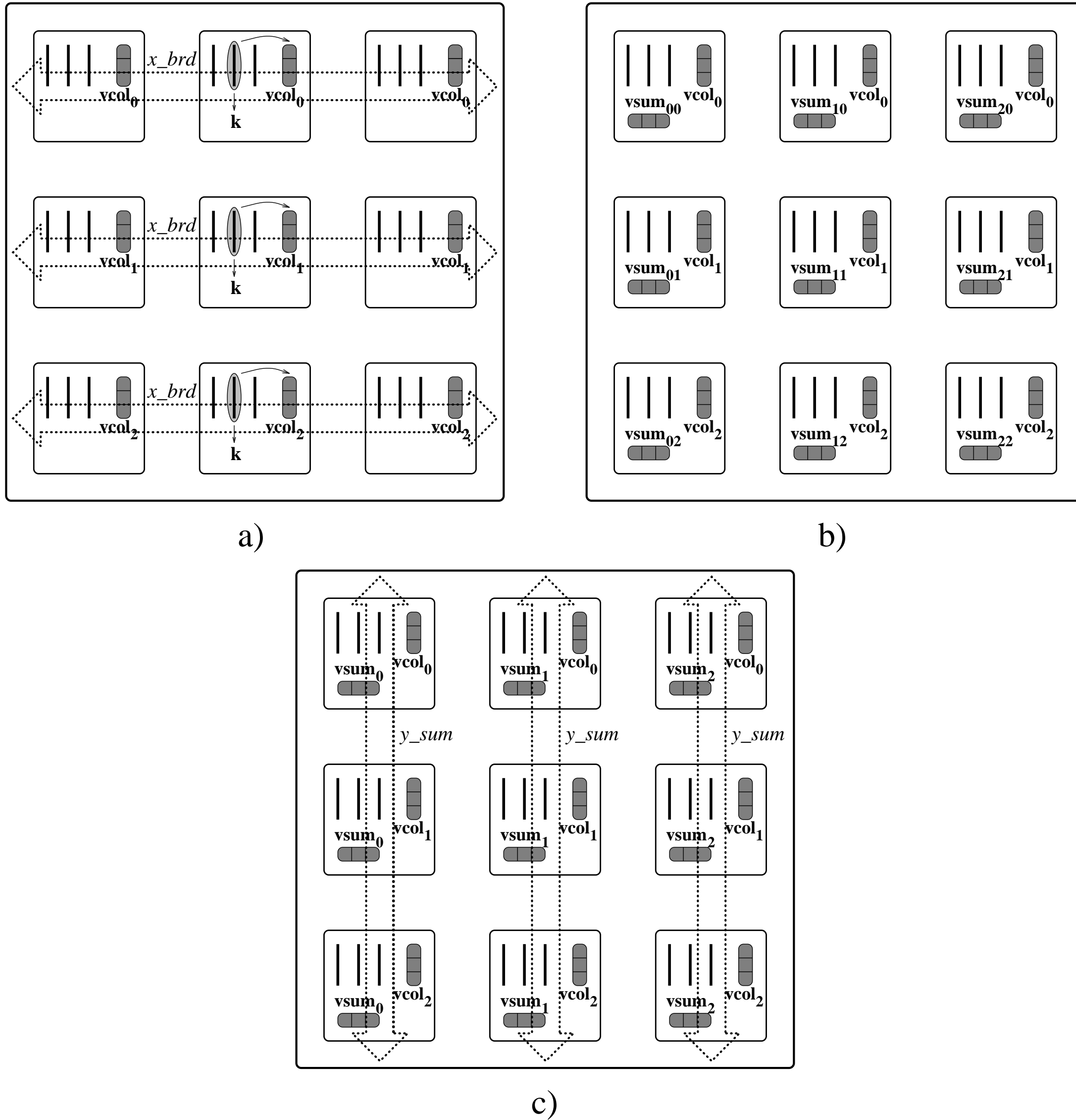

Figure 3.4: *Broadcast and reduction in the parallel MGS procedure*

the matrix (from column k) and an inner loop that traverses only the iterations corresponding to the nonzero elements of vector *vcol*, not the whole iteration space, which saves a lot of computations. When this updating is carried out, we may obtain $a_{ij} = 0$ ($|a_{ij}| < \phi$, to be more precise) and $vcol_i, vsum_j \neq 0$, so that an element of matrix Q which was initially null, now takes a nonzero value (fill-in, see Figure 3.5); this element is, therefore, inserted in the corresponding local list. It is also possible for an element a_{ij} to take a value close to zero, and it must, therefore, be erased from its local list (the opposite phenomenon to fill-in). This section of code is optimized because, each time a column of Q is updated, the corresponding list is traversed only once. The updating of the norms (2.8) is also local, because *vsum* is distributed just as vector *norm*.

The parallel algorithm uses several routines for insertion and deletion in the lists in order to optimize their handling. As an example, there is an *insert* proce-

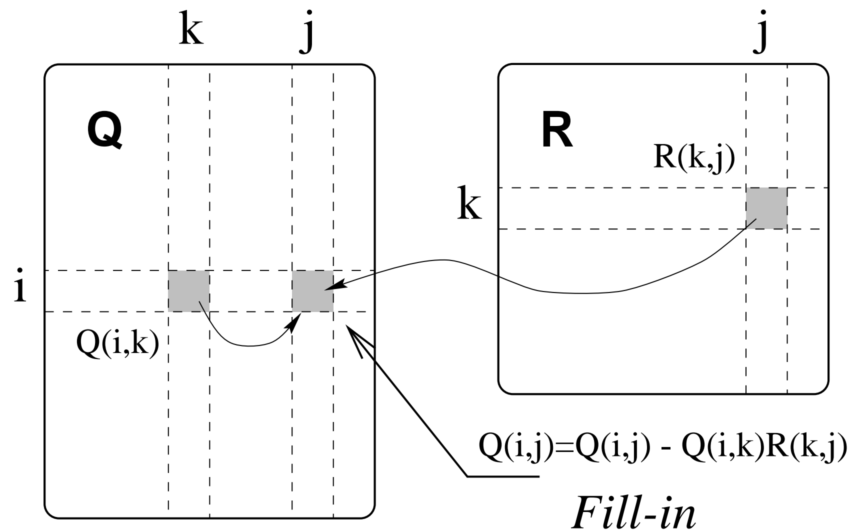

Figure 3.5: *Fill-in in the MGS algorithm*

dure that performs an insertion at the beginning of the list or in the middle and, as a complement, there is another routine *append* that only takes into account insertions at the end of the list. This is due to the fact that in certain steps of the algorithm only insertions at the end of the lists are needed and, this way, computation times can be reduced. For instance, the code is optimized because when in a given iteration k, an insertion at the end of a list is performed due to the fill-in previously described, all the rest of fill-in insertions in that column (list) are also at the end (*append* routine).

We have focused on a linked list implementation of the algorithm, but we have also implemented this algorithm using a data structure for the matrices based on packed vectors (see Figure 2.8). The advantages of this data structure were commented in Section 2.6. The treatment of fill-in with this structure is based on re-copying each processed (and therefore, updated) column. Thus, the entries of each column are processed in growing order of the row index by following this procedure: an auxiliary buffer (which is also a packed vector) with enough size to store one dense column is allocated; during the updating process of a column of the matrix (2.9), each nonzero entry, a previously existing one or a new one (that is, if there is fill-in), is stored in the auxiliary buffer, instead of in the original packed vector. This way, the new elements are just added to the buffer, but the zeroing of existing entries is discarded. After finishing the column updating process, the buffer contains the new packed column. Hence, this auxiliary buffer is just reallocated (*realloc* C function) to a memory block of its exact size, becoming the new column of the sparse matrix, whereas the memory space of the old column is freed. As we can see, memory allocations and releases are performed once every processed column, whereas by using linked lists these operations are performed once every inserted or erased element; so, memory fragmentation generated by the fill-in is reduced to a great extent using packed vectors.

3.3 Parallel Householder Reflections

Parallel Householder-based algorithms are found in [108], implemented on the *Fujitsu AP1000*; Hendrickson [53] presents a Househoulder algorithm without column pivoting on a 1024-processor *nCUBE 2*; and Bischof [14] describes a block Householder factorization on an *Intel iPSC/1* hypercube. All these algorithms are for dense matrices. Regarding sparse factorizations, Raghavan [81] presents a distributed multifrontal row-oriented Householder algorithm on an *Intel iPSC/860* and Matstoms [66] describes a sparse Householder algorithm for shared memory MIMD machines.

We present the parallelization of the sparse column-Householder algorithm in [31]. The calculation of the norms in parallel, as well as the column pivoting have the same considerations as in the MGS algorithm, but taking into account that the swap only involves the coefficient matrix A, which is being upper triangularized (matrix R), and the corresponding norms.

Focusing on the Householder reflection, in order to get the Householder vector v, only column access is required; the current subcolumn $a_{k+1:m-1,k}$ (which was the pivot column before swapping) is divided by a given value (see Figure 2.4, expression (2.10)) and stored in v (previously, we set $v_k = 1$). So, only the column of PEs that contains the current subcolumn k is working in parallel. As the Householder vector contains many null elements and is broadcast to the corresponding PEs on the X axis (*x_brd*), it is stored in a packed vector. Once this is completed, the global entry I of vector v is replicated in the row of PEs with $pidy = I \ mod \ npey$. As we can see, vector v is distributed and replicated in the same way as vector $vcol$ of the MGS procedure.

The BCS-like data distribution scheme (see Section 3.1 and Figure 3.2) we have used for the matrix, allows us to follow an optimal path through submatrix S (described in Section 2.3). The strategy consists in traversing the columns of this submatrix, starting from the last column; when a column whose global index is less than k is reached, the process ends. Also, in order to process each column of the sparse submatrix S, the corresponding list (or packed vector, depending on the implementation) is traversed, starting from the end (the list is doubly-linked) and continuing until a row whose global index is less than k is reached.

Let us be more specific: first, the value β (2.11) is obtained in all the PEs, due to the fact that v is replicated, and using a *y_sum* (sum in the Y axis) reduction. After this, vector $w_{k,n-1} = \beta S^T v$ (2.12) is calculated, so that the global entry J of vector w is replicated in the column of PEs identified by $pidx = J \ mod \ npex$ (the same as vector $vsum$ of the MGS factorization). Note that it is not necessary to transpose submatrix S due to the data distribution scheme; as we can see, the calculation of w is, basically, a sparse matrix (S)-sparse vector (v) product; this fact saves many computations. A *y_sum* reduction is also required to obtain vector w. Finally, submatrix S is updated as $a_{ij} = a_{ij} + v_i \times w_j$ (2.13). As in MGS, it is a local operation performed fully in parallel by all the processors.

Fill-in may occur at this stage if $a_{ij} = 0$ and $v_i, w_j \neq 0$. The updating of the norms is also a local operation (2.14). An auxiliary array of pointers, pointing to row k in the lists or packed vectors (which store matrix A) is required to perform this operation.

All the optimizations in the management of linked lists of the MGS procedure (Section 3.2) are also applicable to the Householder algorithm, as well as the packed vector implementation based on the reallocation of the columns of the matrix.

3.4 Parallel Givens Rotations

A parallel algorithm for dense matrices based on Givens rotations (specifically, fast Givens rotations) on a massively parallel SIMD computer (the *Connection Machine CM-200*) was implemented by Bendtsen et al. [12]. Parallel sparse Givens algorithms are found in [75], for an *SGI Power Challenge* computer, and in [61], where Kratzer implements parallel Givens rotations on a *CM-2* SIMD computer with 16K processors, using a dataflow graph and only nearest-neighbor communications.

We present parallel sparse row-oriented Givens rotations in [98]. The core of Givens algorithm changes substantially with respect to the other two QR algorithms. This is because our algorithm requires access both by rows and by columns: row access because the algorithm updates the entries row by row, as shown in Figure 2.7; and column access to perform the column pivoting. A data structure such as a two-dimensional linked list would be suitable to store matrix A, but it is very costly to manage and requires a great amount of memory space, as we discussed in Section 2.6. Therefore, we use one-dimensional doubly-linked lists (as in MGS and Householder algorithms) to provide access by columns. Efficient row access is achieved using an auxiliary array of pointers (of size n) which points to a certain row of the matrix in the list structure. With this array of pointers, we traverse the linked lists (which represent columns of the matrix) from bottom to top to point to one certain row of the matrix (row access). Figure 3.6 shows an example of this auxiliary array (called *auxp*), pointing to the row 3 of the sparse matrix of Figure 2.8. If the corresponding element of this row is zero (as it occurs in the second and third columns), *auxp* points to the previous element of the list because, in this algorithm, we traverse the lists in the reverse order. Therefore, the entries of the array *auxp* are updated in each iteration of the loop i of expression (2.15), in order to make them point to the previous row $(i-1)$ of the sparse matrix.

Let us consider the Givens sequential algorithm (Figure 2.6) to analyze how it can be executed in parallel. If these rotations are parallelized according to the sequential algorithm, neither the outer loop k, which traverses the columns of the matrix, nor the inner loop (2.15), which annihilates each element of the column, could be executed in parallel due to data dependences. Thus, the parallel algorithm would need n(2m-n-1)/2 iterations (in the dense case). Nevertheless, a

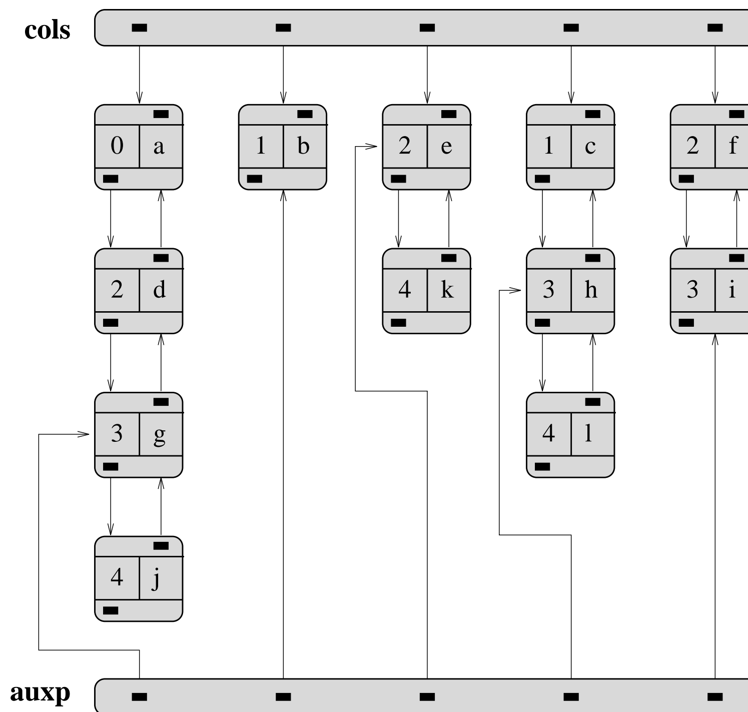

Figure 3.6: *Row access in a column-oriented data structure*

Givens rotation (2.17) can be applied to any two rows, not necessarily adjacent; therefore, each row of PEs can apply, independently and in parallel, the rotations to the rows of A which they store, so that execution times are reduced. Communications are required to broadcast *gcos* and *gsin* to the corresponding rows of PEs (*x_brd*).

This process is illustrated for a *32*×*4* matrix distributed on an *8*×*1* mesh (Figure 3.7i). In the example of Figure 3.7, the eight PEs apply, in parallel, Givens rotations to the rows of the first column of the matrix, but the first row of each PE is not rotated. Rows which have not been rotated are paired 2 by 2, in parallel. This implies $\lceil log_2\ npey \rceil$ steps at most. In this example, $npey = 8$ and three steps are required: 3.7ii, 3.7iii and 3.7iv; the final result is shown in Figure 3.7v. The number of steps can be reduced in some cases; for instance, some PEs may not have elements to be annihilated, due to the sparsity of the matrix and/or due to the fact that the algorithm is computing the last iterations, which reduces communications. In the previous example, if 4 PEs did not have any element to be zeroed, only $\lceil log_2\ 8\text{-}4 \rceil = 2$ steps would be necessary. The procedure shown in Figure 3.7 was generalized in our parallel Givens algorithm for any value of *npey*, and was not restricted to powers of 2.

Rows are rotated to annihilate the corresponding entries. It is clear that, with the sparse approach, those rows whose first element is zero do not need to be rotated. As our matrix is sparse and there are no null elements stored in the lists, we traverse the entries of list (column) k and rotate only the rows of nonzero elements. This way, we save many rotations, and running times are decreased. Obviously, fill-in may appear at this stage, when applying a rotation following

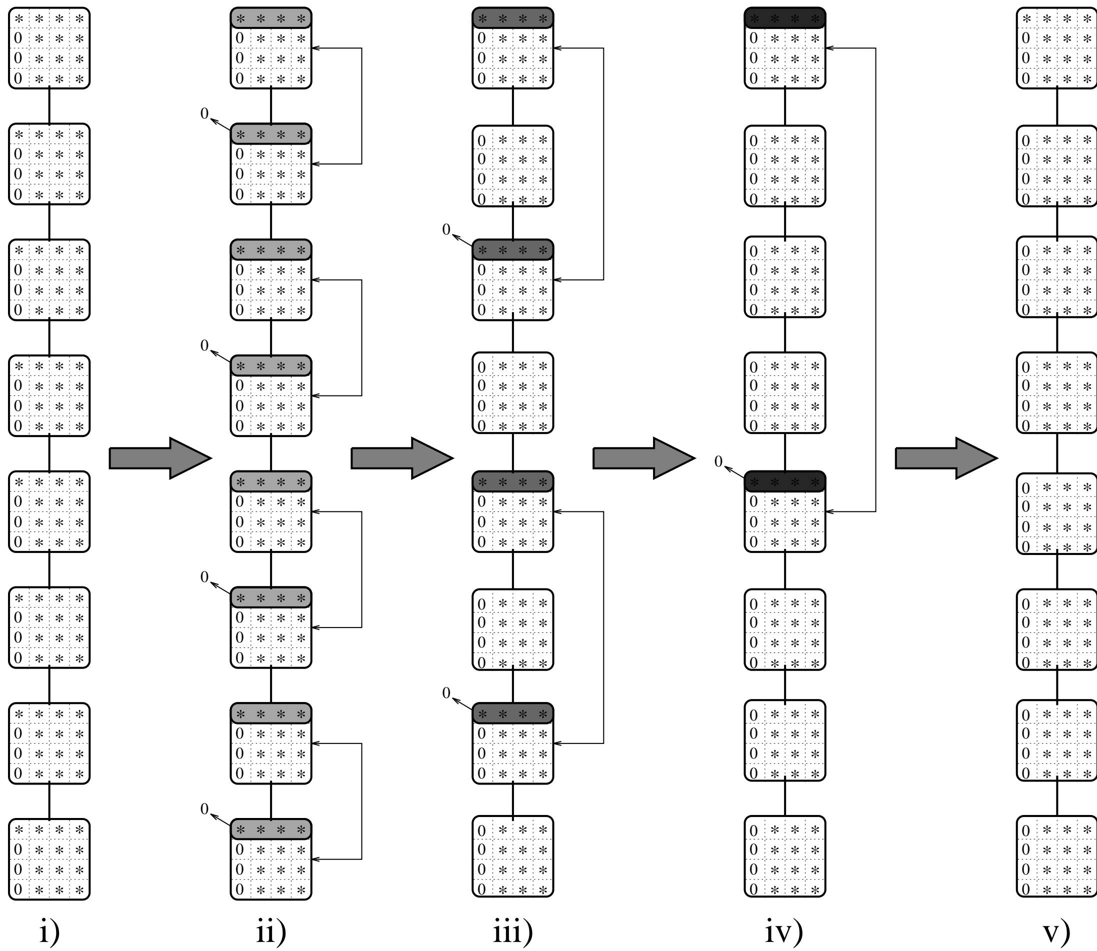

Figure 3.7: *Parallel Givens rotations*

expression (2.17). The parallel algorithm uses several routines for insertion and deletion in the lists in order to improve their management.

The parallel approach of the column pivoting procedure, the initial norm calculation and norm updating (2.16) are similar to the ones of the Householder algorithm.

3.5 Parallel Least Squares

As we said in Section 2.5, the least squares problem $min \; \|Ax - b\|_2$ can be approached by means of QR factorization, by solving the upper triangular system $R\Pi^T x = Q^T b$ in three stages: calculation of $Q^T b$, back-substitution and permutation.

The product $Q^T b$ is calculated at the same time as the QR factorization is performed. Previously, in MGS, vector b (we assume it is dense) was distributed in each column of PEs, so that the global component I of b is replicated in the row of PEs with *pidy*=*I mod npey*. The vector named *qtb* contains the product

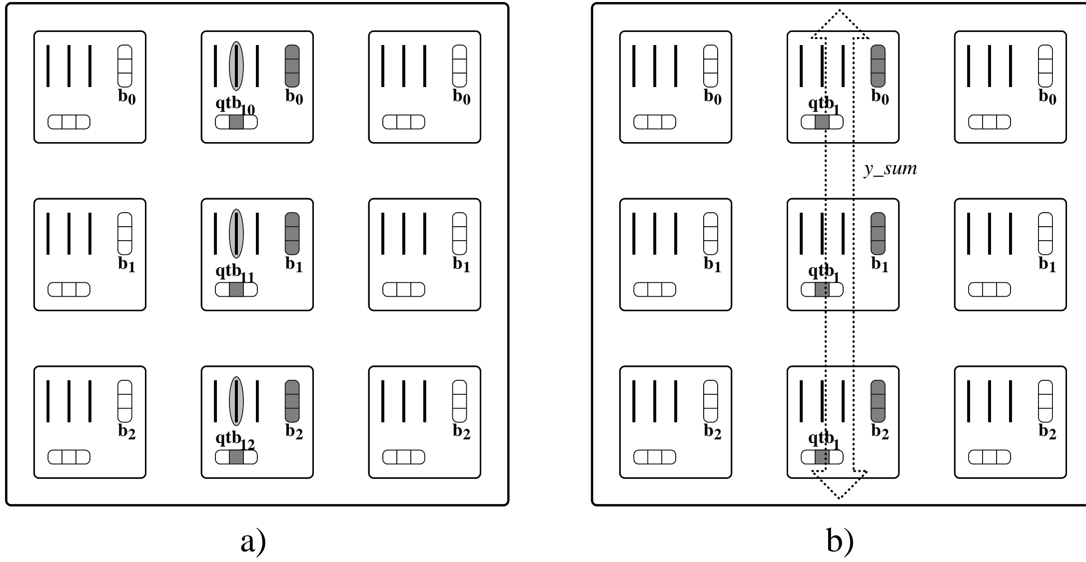

Figure 3.8: *Calculation of Q^Tb*

Q^Tb, so that one element of *qtb* is obtained for each iteration k of the algorithm. Each element is a dot product between vector b and the column of Q obtained in that iteration (obviously, due to the data distribution, it is not necessary to transpose matrix Q). Figure 3.8a shows this situation, where the current column of Q being processed (iteration k) is the middle one, and the local dot product is calculated. Then, in Figure 3.8b the global dot product is obtained by means of a reduction instruction (*y_sum*). In order to group communications, this reduction is performed at the end of the factorization, for the whole vector *qtb* instead of entry by entry. We describe the parallelization of the least squares problem through the MGS algorithm in [30].

Matrix Q is not explicitly calculated by means of Householder reflections. In order to get Q^Tb, vector b is stored in vector *qtb* (note that, unlike MGS, *qtb* becomes a replicated column vector). And, in each one of the iterations (n, if the rank is maximum) of our parallel algorithm, a Householder transformation is applied to elements $qtb_{k:m-1}$: $qtb = qtb + v\lambda$ (similar to (2.13)), where $\lambda = \beta qtb^T v$ (similar to (2.12)), is a real number. Only a reduction *y_sum* is necessary to obtain λ in each iteration. The rest of the operations are local. As we can see, the required entries of *qtb* ($qtb_{0:n-1}$) are obtained in-place over vector b.

Using Givens rotations, vector b is stored in vector *qtb*, in the same way as in Householder reflections (in-place procedure). Next, Givens rotations are applied to the corresponding elements of *qtb* at the same time as the rotations to the rows of matrix A are applied. For instance, if rows α and β are being processed, this product is calculated as follows (it is similar to (2.17)):

$$\begin{pmatrix} qtb'_\alpha \\ qtb'_\beta \end{pmatrix} \leftarrow \begin{pmatrix} gcos & -gsin \\ gsin & gcos \end{pmatrix} \cdot \begin{pmatrix} qtb_\alpha \\ qtb_\beta \end{pmatrix} \tag{3.1}$$

It is a local operation, as *gcos, gsin* were previously broadcast to the corresponding processors, as described in Section 3.4.

Once $Q^T b$ is calculated, the upper triangular system $Rx = Q^T b$ is solved by means of a back-substitution. The corresponding sequential algorithm is as follows:

$$\begin{aligned} & \textit{for } (i{=}rank{-}1; i \geq 0; i{-}{-}) \\ & \quad x_i = (qtb_i - \sum_{j=i+1}^{rank-1} r_{ij} \cdot x_j)/r_{ii}; \end{aligned} \tag{3.2}$$

This loop has data dependences, and thus it must be kept in the parallel code without any possibility of being distributed among the PEs. However, it is a limitation for dense systems. Cabaleiro and Pena [19] propose a parallel back-substitution by taking advantage of the sparse nature of the system. They implemented a data-driven back-substitution, making use of non-blocking communications in order to dynamically extract the inherent parallelism of sparse systems. This strategy is very interesting for algorithms in which the back or forward substitutions represent an important part of the whole running time of the application (for instance, the incomplete LU factorization [10]). This is not the case for our QR factorizations and, therefore, this approach is not worth the trouble of implementing it due to its complexity.

In addition, it is necessary to access the entries of matrix R by rows (matrix R is stored by columns). There are several options to solve this drawback. One of them is to apply the column version of the back-substitution, described in [42, Chapter 3]. Another option is to change the storage scheme of matrix R to a row-oriented one. The solution we have implemented is to maintain the storage scheme of matrix R and use an auxiliary pointer vector to allow row access, in the same way as shown in Figure 3.6 and described in Section 3.4 for Givens rotations.

The communications required for the back-substitution are: a reduction operation (*x_sum*, sum by rows) to obtain the component $\sum_{j=i+1}^{rank-1} r_{ij} \cdot x_j$ of expression (3.2); and a broadcast operation (*y_brd*, broadcast by columns) to send every entry of the solution vector x, as it is being obtained, to the corresponding column of PEs.

Once the back-substitution is carried out we get the solution vector $x \in \Re^n$ distributed cyclically on each row of PEs, so that the global entry J of vector x is replicated in the column of PEs with $pidx = J \ mod \ npex$.

The last phase of the least squares problem is the permutation stage. Due to the column pivoting carried out in the QR factorization, the Π permutation must be applied to the components of vector x, so that x is overwritten with vector Πx. All the PEs contain a vector named *permute* $\in \Re^n$. It is the only vector whose components are not distributed among the PEs (all the PEs contain the full vector replicated). This vector stores the index of the column swapped in

each iteration (p) and, by applying these swaps to the components of x starting from the end, the entries of x are obtained in the correct order. This permutation operation is inherently sequential, but it means a minimal part in the execution time of the whole factorization procedure.

Chapter 4

Experimental Results

4.1 Parallel Machines

The supercomputer in which the parallel algorithms were originally developed was the *Fujitsu AP1000* MIMD distributed memory computer, which is composed of a host computer and from 64 to 1024 processing elements or cells (SPARC processors) in a two-dimensional torus topology.

It also has three independent communication networks, depicted in Figure 4.1: the *torus* network (*T-net*, with hardware support for wormhole routing), for point to point communications between cells; the *broadcast* network (*B-net*), for 1 to N communications between the host and the cells, as well as for data distribution and collection; and the *synchronization* network (*S-net*), for the barrier synchronization (consult [59] for more details on the architecture). It has its own set of native message-passing routines for parallel processing [38]. This environment, which is called CellOS, includes gather/scatter instructions, reduction instructions, broadcast and messaging routines, etc. We used this set of instructions for our parallel algorithms, although the standard message-passing library MPI [71][72] was later supported by the AP1000. The AP1000 parallel programming environment is extensively detailed in [55]. The parallel QR algorithms were debugged and tested on the software AP1000 simulator CASIM [39].

All the algorithms were also implemented on the *Cray T3D* distributed memory multiprocessor, shown in Figure 4.2. It has a *Cray Y-MP* as server, and from 32 to 2048 DEC-Alpha 21064 processors at 150 MHz, connected by a 3-D torus topology. This architecture, its programming models and programming tools are described in detail in [25]. The PVM message-passing library [40][92], due to portability reasons, was selected to program our parallel algorithms on the Cray T3D. We have also used low latency communication functions, such as *pvm_fastsend* and *pvm_fastrecv* (non-standard PVM functions) for messages of size less than 256 bytes. We have also developed specific reduction and broadcast PVM-based routines suitable for our algorithms in order to group together communications insofar as this has been possible. The parallel algorithms for

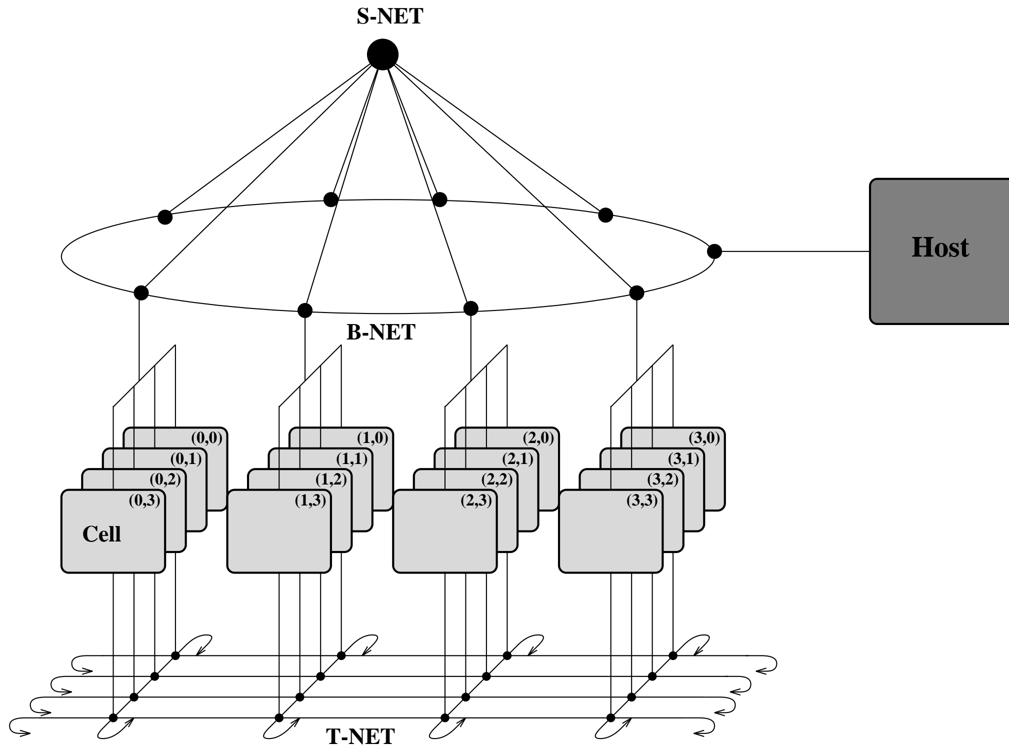

Figure 4.1: *Architecture of the Fujitsu AP1000*

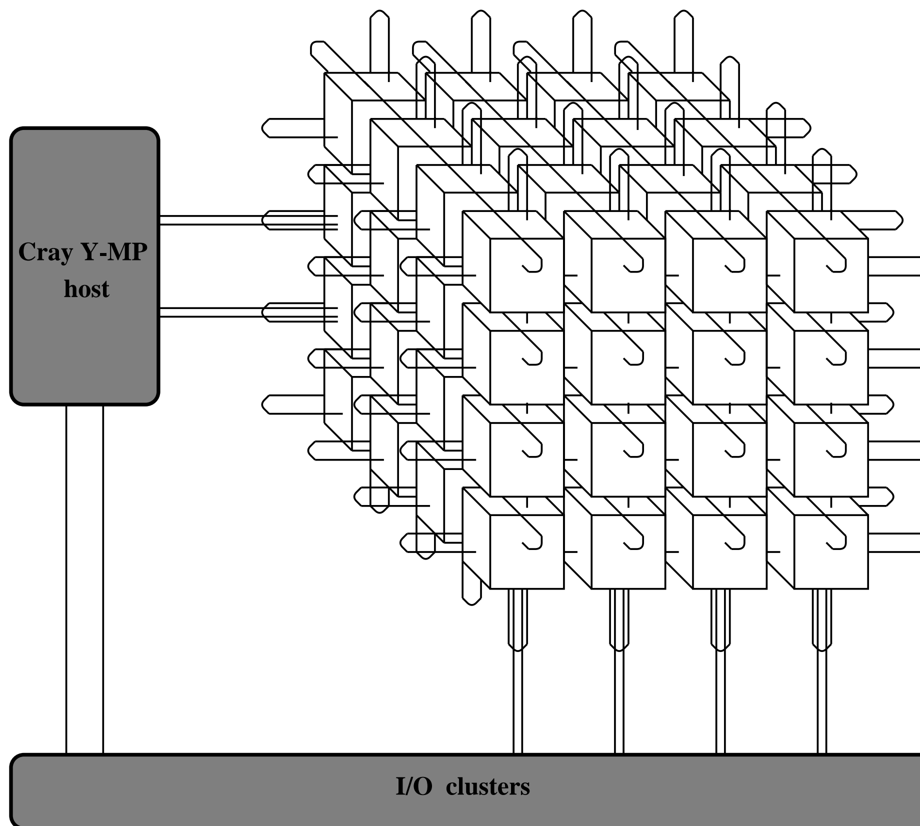

Figure 4.2: *Architecture of the Cray T3D*

	Fujitsu AP1000	**Cray T3D**	**Cray T3E**
Processor	SPARC	DEC Alpha 21064 (150 Mhz)	DEC Alpha 21164 (300 Mhz)
Processor peak performance	8.33 MFLOPS (single precision) 5.67 MFLOPS (double precision)	150 MFLOPS	600 MFLOPS
Local memory	16 Mb	64 Mb	64 Mb to 2 Gb
#processors	64 to 1024	32 to 2048	16 to 2048
Peak interprocessor communication rate	50 Mb/s (B-net) 25 Mb/s (T-net)	300 Mb/s (3D torus)	480 Mb/s (3D torus)

Table 4.1: *Characteristics of the target supercomputers*

this machine were previously debugged on a cluster of workstations in a PVM environment.

The parallel codes, for both supercomputers, were written in C language, using double precision floating point numbers and following the SPMD (Simple Program Multiple Data) paradigm.

Table 4.1 summarizes the characteristics of the AP1000 and T3D multiprocessors. Obviously, the peak parameters are achieved under ideal conditions. For instance, the routing hardware of the AP1000 T-net provides a theoretical interprocessor communication rate of 25 Mb/s between any two cells but, in practice, about 6 Mb/s is attainable by user programs.

The Cray T3E supercomputer was also included in this table, with comparison purposes, because it will be the target machine of further experiments described in Chapter 7.

4.2 Execution Times and Efficiencies

All the experimental results presented in this chapter were obtained by setting $\epsilon = 1$ in the pivoting strategy of expression (2.18), in order to reduce fill-in as much as possible.

Tables 4.2, 4.3, 4.4 present the execution times (in seconds) and efficiencies on the AP1000 of the QR algorithms, for MGS, Householder and Givens, respectively. The use of square meshes is not a restriction, rectangular meshes could be employed. The execution times include QR factorization as well as the solving of the least squares problem. The time required for data distribution and collection is not included because we assume that these algorithms are possible subproblems within larger programs. Three sparse matrices from the Harwell-Boeing set (see Table 2.1) have been chosen: *JPWH991*, *SHERMAN1* and *MAHINDAS*.

Matrix	*1 PE*		*2×2 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	2415.48s	2152.33s	646.37s (0.93)	576.76s (0.93)
SHERMAN1	510.20s	451.42s	159.39s (0.80)	143.99s (0.78)
MAHINDAS	246.94s	221.39s	83.24s (0.74)	77.05s (0.72)
	4×4 PEs		*8×8 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	183.68s (0.82)	164.02s (0.82)	54.02s (0.70)	50.31s (0.67)
SHERMAN1	47.56s (0.67)	44.60s (0.63)	15.91s (0.50)	15.03s (0.47)
MAHINDAS	31.83s (0.48)	30.50s (0.45)	14.56s (0.27)	13.87s (0.25)

Table 4.2: *MGS: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)*

Matrix	*1 PE*		*2×2 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	2283.24s	2093.75s	596.71s (0.96)	553.20s (0.95)
SHERMAN1	645.35s	561.04s	190.02s (0.85)	167.47s (0.84)
MAHINDAS	277.98s	251.16s	91.65s (0.76)	87.90s (0.71)
	4×4 PEs		*8×8 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	162.44s (0.88)	155.64s (0.84)	46.45s (0.77)	44.97s (0.73)
SHERMAN1	53.59s (0.75)	50.07s (0.70)	17.89s (0.56)	17.58s (0.50)
MAHINDAS	33.91s (0.51)	33.44s (0.47)	15.46s (0.28)	15.37s (0.26)

Table 4.3: *Householder: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists (DLL) and packed vectors (PV)*

Matrix	*1 PE*	*2×2 PEs*	*4×4 PEs*	*8×8 PEs*
JPWH991	1089.05s	327.60s (0.83)	91.87s (0.74)	35.83s (0.47)
SHERMAN1	221.04s	90.41s (0.61)	25.62s (0.54)	11.67s (0.30)
MAHINDAS	140.39s	63.13s (0.56)	23.11s (0.38)	10.72s (0.20)

Table 4.4: *Givens: execution times and efficiencies on the AP1000 for* $\epsilon = 1$*, using doubly-linked lists*

Matrix	*1 PE*		*2×2 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	241.36s	202.50s	71.35s (0.85)	59.61s (0.85)
ORANI678	504.42s	384.80s	146.09s (0.86)	114.46s (0.84)
SHERMAN5	607.35s	512.27s	172.62s (0.88)	149.42s (0.86)
	4×4 PEs		*8×8 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	18.98s (0.79)	16.63s (0.76)	7.43s (0.51)	7.29s (0.43)
ORANI678	50.30s (0.63)	41.50s (0.58)	21.56s (0.37)	21.03s (0.29)
SHERMAN5	73.62s (0.52)	64.61s (0.50)	26.96s (0.35)	26.95s (0.30)

Table 4.5: *MGS: execution times and efficiencies on the Cray T3D for $\epsilon = 1$, using doubly-linked lists (DLL) and packed vectors (PV)*

Matrix	*1 PE*		*2×2 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	232.45s	161.02s	63.09s (0.92)	46.31s (0.87)
ORANI678	765.25s	594.51s	249.33s (0.77)	200.80s (0.74)
SHERMAN5	1156.79s	890.67s	346.44s (0.83)	275.37s (0.81)
	4×4 PEs		*8×8 PEs*	
	DLL	**PV**	**DLL**	**PV**
JPWH991	19.10s (0.76)	15.81s (0.64)	8.39s (0.43)	8.13s (0.31)
ORANI678	70.53s (0.68)	60.11s (0.62)	29.97s (0.40)	28.94s (0.32)
SHERMAN5	99.29s (0.73)	85.85s (0.65)	42.64s (0.42)	40.48s (0.34)

Table 4.6: *Householder: execution times and efficiencies on the Cray T3D for $\epsilon = 1$, using doubly-linked lists (DLL) and packed vectors (PV)*

Matrix	*1 PE*	*2×2 PEs*	*4×4 PEs*	*8×8 PEs*
JPWH991	148.10s	48.86s (0.76)	16.40s (0.56)	7.67s (0.30)
ORANI678	288.43s	95.50s (0.76)	37.66s (0.48)	18.59s (0.24)
SHERMAN5	452.64s	143.88s (0.79)	59.99s (0.47)	32.22s (0.22)

Table 4.7: *Givens: execution times and efficiencies on the Cray T3D for $\epsilon = 1$, using doubly-linked lists*

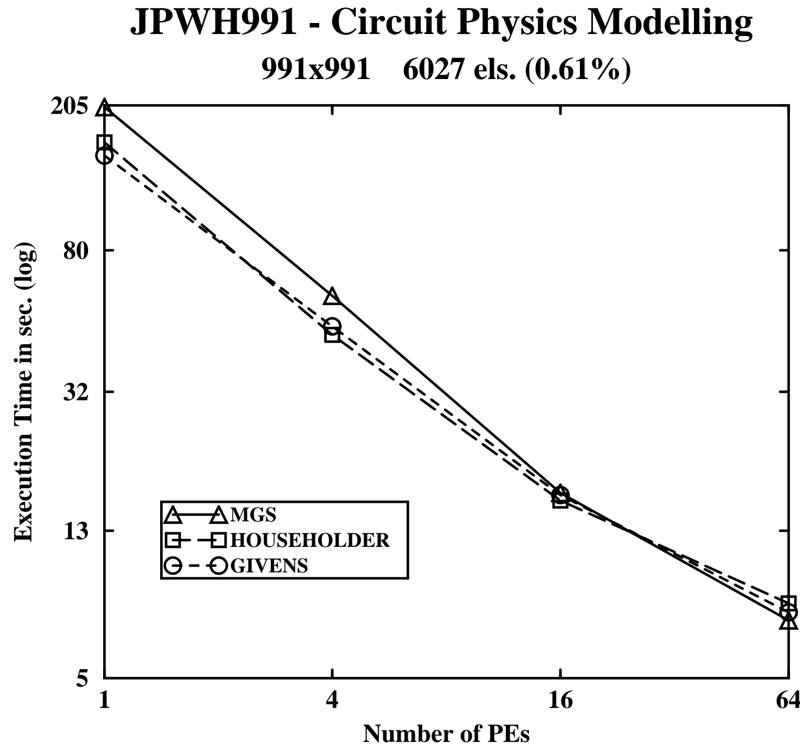

Figure 4.3: *Matrix JPWH991: execution times for the QR algorithms on the Cray T3D*

Tables 4.5, 4.6, 4.7 show the same information for the Harwell-Boeing matrices *JPWH991*, *ORANI678* and *SHERMAN5*, which have enough fill-in and are large enough to be executed on a multiprocessor like the Cray T3D. Matrix *JPWH991* was intentionally included in the experiments of both supercomputers in order to have a comparative reference on the computing power of the AP1000 and the Cray T3D machines. As observed in the execution times of this matrix for one PE, the execution on the AP1000 SPARC PE takes approximately one order of magnitude more than on the T3D Alpha PE. The remaining matrix in Table 2.1 (*SHL400*) was discarded in these experiments because the computation time for its factorization is not large enough to be worth being executed in these parallel computers.

In the case of MGS and Householder, execution times are provided using doubly-linked lists and packed vectors as data structures. The advantages of packed vectors as opposed to linked lists were detailed in Section 2.6. Basically, as packed vectors store the nonzero entries of the matrix in adjacent memory positions, cache misses are reduced to a great extent and, therefore, execution times are decreased, as shown experimentally in Tables 4.2, 4.3, 4.5 and 4.6. This fact is very clear for small meshes but, as can be observed, execution times using doubly-linked lists and packed vectors are closer and closer as the number of PEs increases because communications are the same for both schemes and there are fewer local computations. As a consequence of that, efficiencies are worse using packed vectors.

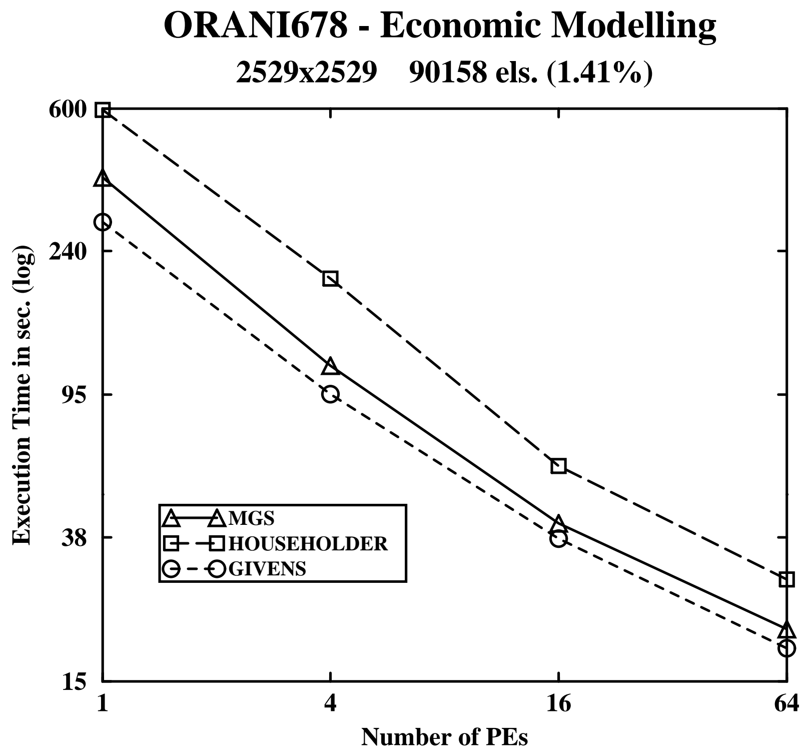

Figure 4.4: *Matrix ORANI678: execution times for the QR algorithms on the Cray T3D*

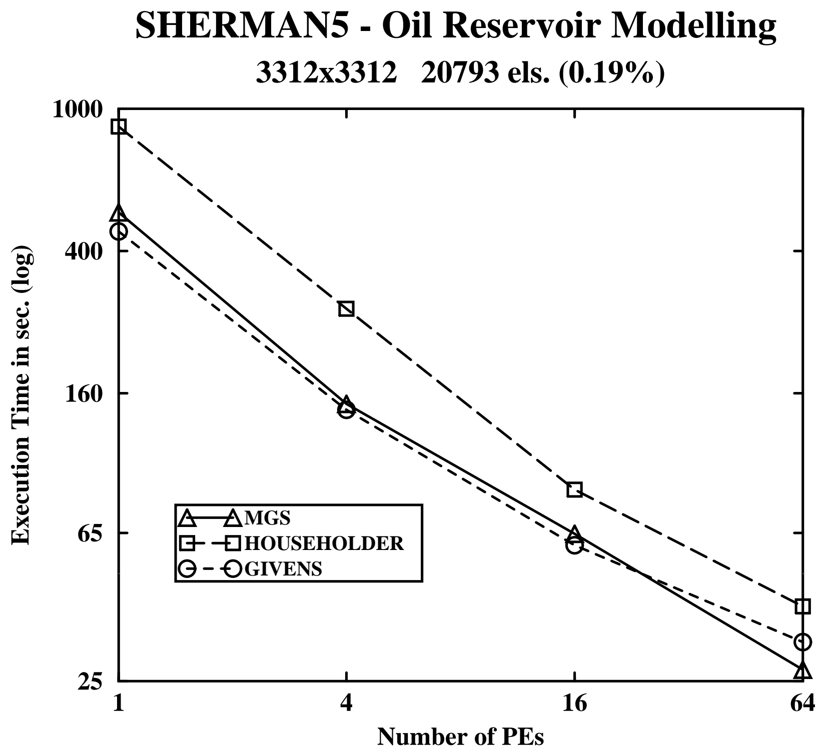

Figure 4.5: *Matrix SHERMAN5: execution times for the QR algorithms on the Cray T3D*

As a comparison of the three algorithms, the Householder factorization is, in general, the one which presents the highest running times (except in matrix *JPWH991*); this is due to the great amount of fill-in which appears in matrix R during Householder factorization (see next section, Table 4.8), which increases the number of computations with the new nonzero entries. On the other hand, the fastest algorithm is the one based in Givens rotations (even compared to the MGS and Householder algorithms implemented with packed vectors). As we can see, the running times of the three algorithms come closer as the number of PEs increases, specially for MGS and Givens (it is clear for an 8×8 mesh) because, when data are distributed among more PEs, the number of computations that each processor carries out is smaller and the communication term is a relatively more significant fraction of the running time. Figures 4.3, 4.4 and 4.5 present a graphical comparison among the execution times of the three QR algorithms on the Cray T3D, for matrices *JPWH991*, *ORANI678* and *SHERMAN5*, respectively. In these figures, the execution times for MGS and Householder are the ones obtained using packed vectors, while in Givens rotations the only available doubly-linked list running times were used.

The efficiency for $npes$ PEs (E_{npes}) is defined as $E_{npes} = \left(\frac{T_1}{npes \times T_{npes}}\right)$, being T_{npes} the execution time using $npes$ PEs. With regard to the efficiencies, as we can see in the corresponding tables, the algorithms scale rather well. Nevertheless, better efficiencies could be achieved with larger matrices because, in general, there are more calculations and the running time of the local task is large in relation to the additional time required by communications; thus, the parallelism is more efficiently exploited. Efficiencies are worse for the Givens algorithm, not only for having lower execution times than the other algorithms, but also for the number of communications which are required to update the non-rotated row of each PE in each step of the Givens factorization (see Figure 3.7). The influence of this factor increases for large mesh sizes; for instance, note that for matrices *JPWH991* and *SHERMAN5* the running times on the Cray T3D are lower in Givens than in MGS for all the mesh configurations, except for an 8×8 mesh topology, in which the execution times are higher in Givens than in MGS. This results in lower efficiencies.

As a last remark, it is obvious that efficiencies are much better in the AP1000 than in the Cray T3D (see matrix *JPWH991*) because the ratio *Processor peak performance/Peak interprocessor communication rate* (see Table 4.1) is much lower in the AP1000 than in the T3D. That is, the interprocessor network in the AP1000 is very fast in relation to the SPARC processor, which results in good efficiencies.

4.3 Fill-in and Numerical Results

Let us analyze how the parallel execution of the algorithms affects the fill-in of matrix R and numerical stability. As shown in Table 4.8, sparsity fluctuates

Matrix	*1 PE*			*2×2 PEs*		
	MGS	**HOU**	**GIV**	**MGS**	**HOU**	**GIV**
SHL400	1712	1712	1712	1712	1712	1712
JPWH991	174307	262486	174930	184806	262105	183583
SHERMAN1	53560	104238	51552	53560	102636	56290
MAHINDAS	32392	61412	34157	32392	61220	33785
ORANI678	354720	516293	356227	352686	547765	355601
SHERMAN5	343218	850164	343635	350747	866825	310017
	4×4 PEs			*8×8 PEs*		
	MGS	**HOU**	**GIV**	**MGS**	**HOU**	**GIV**
SHL400	1712	1712	1712	1712	1712	1712
JPWH991	171019	263295	167191	161431	254009	141080
SHERMAN1	53560	103176	43677	53560	102227	53706
MAHINDAS	32392	62471	33715	32392	62528	33672
ORANI678	353855	534198	351486	353473	535828	355489
SHERMAN5	375562	861524	301554	343352	878523	341842

Table 4.8: *Number of nonzero entries in matrix R using several mesh configurations* ($\epsilon = 1$)

Matrix	*1 PE*			*2×2 PEs*		
	MGS	**HOU**	**GIV**	**MGS**	**HOU**	**GIV**
SHL400	0	0	0	0	0	0
JPWH991	9.99E-15	1.44E-14	2.22E-14	8.66E-15	1.53E-14	1.64E-14
SHERMAN1	3.34E-13	3.43E-12	3.03E-12	2.98E-13	3.39E-12	1.57E-12
MAHINDAS	6.39E-7	1.70E-9	2.22E-9	6.53E-7	3.04E-9	3.91E-10
ORANI678	1.15E-12	1.14E-13	1.10E-13	6.21E-13	8.13E-14	9.28E-14
SHERMAN5	6.48E-9	1.32E-12	5.34E-13	2.32E-9	2.44E-12	5.05E-13
	4×4 PEs			*8×8 PEs*		
	MGS	**HOU**	**GIV**	**MGS**	**HOU**	**GIV**
SHL400	0	0	0	0	0	0
JPWH991	6.88E-15	1.69E-14	1.93E-14	6.88E-15	1.20E-14	1.20E-14
SHERMAN1	1.56E-13	3.48E-12	5.08E-13	6.43E-13	1.28E-12	8.55E-13
MAHINDAS	6.70E-7	1.38E-9	6.15E-10	6.71E-7	2.29E-9	2.76E-9
ORANI678	1.86E-12	1.17E-13	5.20E-14	9.92E-13	8.70E-14	1.14E-13
SHERMAN5	2.38E-9	1.02E-12	7.77E-13	4.17E-10	4.21E-13	8.42E-13

Table 4.9: *Numerical errors using several mesh configurations* ($\epsilon = 1$)

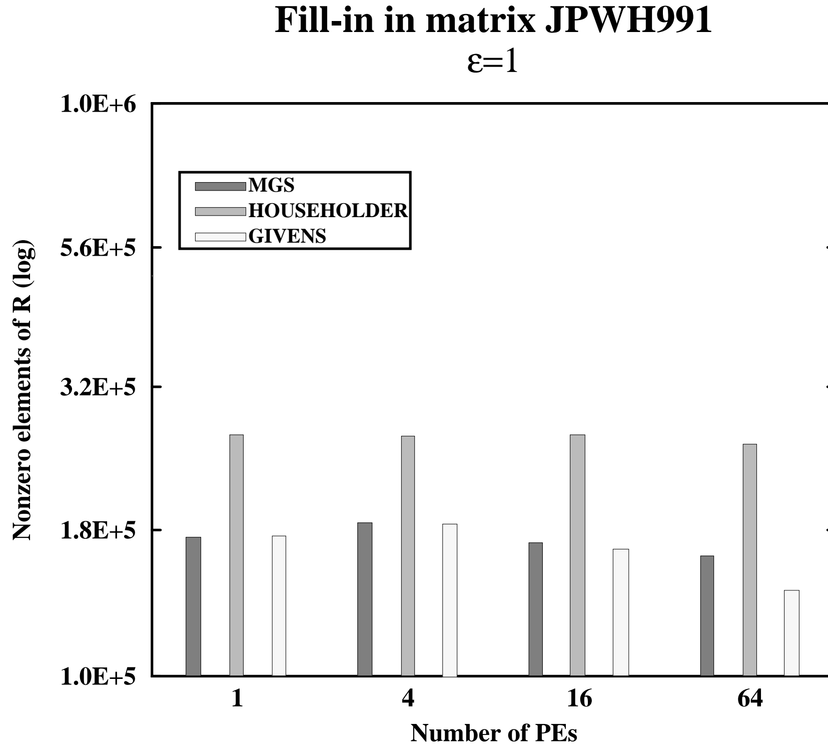

Figure 4.6: *Matrix JPWH991: fill-in in matrix R for different mesh sizes*

depending on the mesh configuration that is used (1, 4, 16 and 64 PEs). The explanation for this lies in the fact that, when choosing the pivot column p following criterion (2.18), a tie between two (or more) candidate columns is possible. Obviously, the pivot column, selected by means of a reduction instruction, varies depending on the mesh configuration, whereupon factorization will be different. This fluctuation in the number of elements in matrix R is greater using Givens rotations since, depending on the configuration of the mesh (more specifically, depending on the number of processors on the Y axis, *npey*), the pairs of rows of the coefficient matrix A involved in each rotation vary (due to the cyclic distribution) and, therefore, fill-in is different. Figures 4.6, 4.7, 4.8 depict the sparsity in matrix R for matrices *JPWH991*, *ORANI678* and *SHERMAN5*, respectively, in order to have a graphical and clear view of part of the results shown in Table 4.8.

It is very important, from the numerical stability point of view of our parallel algorithms that, although fill-in varies, the numerical errors described in Subsection 2.7.2 remain invariant using different mesh sizes. This fact is checked in Table 4.9. As we can see, the numerical errors vary slightly depending on the number of PEs, for the same reason mentioned above for the fill-in. Nevertheless, within each particular algorithm, the variation in error, not following any special pattern or tendency, is not significant for the various mesh configurations. Finally, we present additional experimental results on the AP1000 in a technical report [96].

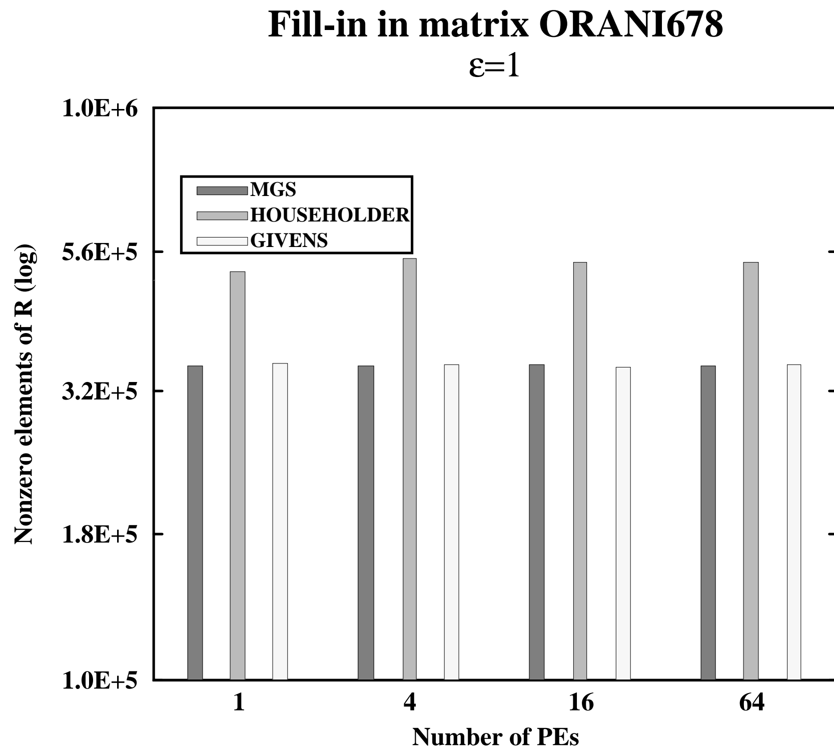

Figure 4.7: *Matrix ORANI678: fill-in in matrix R for different mesh sizes*

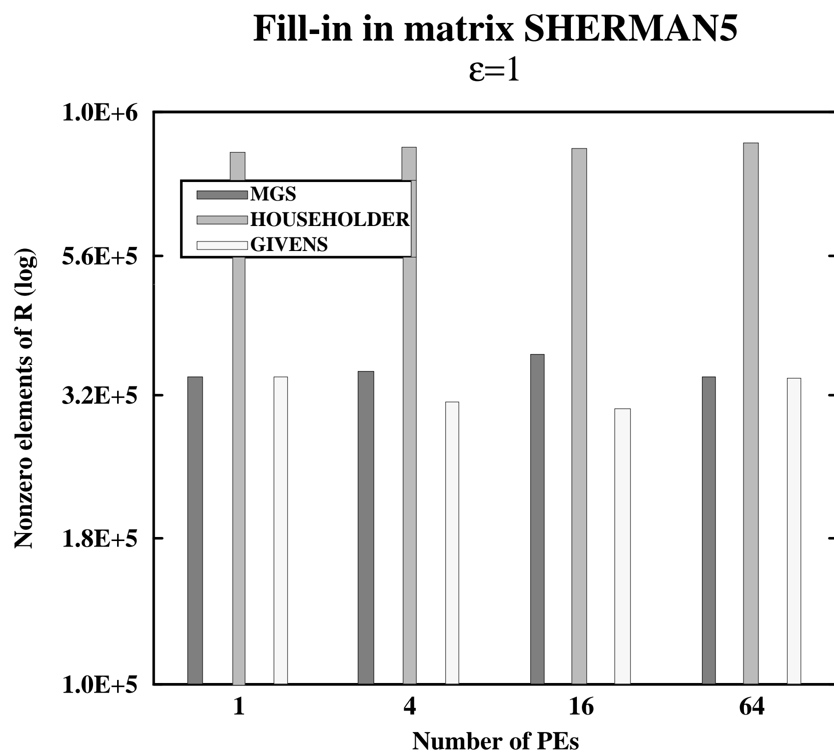

Figure 4.8: *Matrix SHERMAN5: fill-in in matrix R for different mesh sizes*

4.4 Local Pivoting

The criterion to preserve sparsity proposed in Subsection 2.7.1 (expression (2.18)) is a global pivoting strategy, to the effect that the candidate pivot columns may be located in any column of PEs. If the pivot column p and the current column k are located in different processors, communications are necessary to perform the corresponding column swap (explicit pivoting), as described in Section 3.2.

Assuming the equi-probability of each column of processors to contain the selected pivot column following expression (2.18) (this assumption can be made due to the utilization of a cyclic distribution), and assuming, to simplify, that the coefficient matrix to be factorized is maximum-rank, the mean number of communications (data exchanges) required to perform the pivoting is $\left(1 - \frac{1}{npex}\right) \times n$. Obviously, as the X dimension of the mesh ($npex$) increases, a larger number of communications will be necessary because the probability of columns k and p being located in the same column of PEs decreases.

Our proposal to reduce the pivoting communications consists in applying a local pivoting, so that the candidate pivot columns are restricted to those located in the column of PEs which contains column k. Therefore, we should add the condition *'and (k mod npex) = (p mod npex)'* to the criterion (2.18). With this strategy, pivoting is simply reduced to a pointer exchange, and communications are eliminated; it is, therefore, a local operation. Besides, if linked lists are used as storage scheme, we also save the execution times required by the pivoting operation to dump/reconstruct the lists in/from a buffer, as shown in Figure 3.3. It is also important to note that the fact of using local pivoting does not eliminate the rank-revealing condition of the factorization.

Nevertheless, in the last iterations of the QR algorithms, there are few candidate pivot columns in the corresponding column of PEs and, in this situation, it would be advisable to apply global pivoting to ensure numerical stability and sparsity preservation. In order to include this feature, we define a parameter named γ as the number of iterations of the algorithms in which local pivoting is applied divided by the total number of iterations (n if the matrix is maximum-rank). Taking into account the previous assumptions, the mean number of communications required by the pivoting strategy becomes $\left(1 - \frac{1}{npex}\right) \times (1 - \gamma) \times n$. In practice, as we work with large sparse matrices (n high), we use γ values greater than 0.9. Therefore, in the code of Figure 2.1 we should add the following condition at the beginning of loop k: *'if (k< γ n) apply local pivoting'*.

We have also considered those cases in which, for a certain iteration, none of the candidate pivot columns in the corresponding column of PEs satisfy criterion (2.18) (for instance, for all the candidate pivot columns, their norm is less than the threshold ϕ). In this situation, which may happen in any iteration, global pivoting is applied in that iteration, taking advantage of the calculations previously performed in the local pivoting. That is, in fact, all the columns of PEs, not only the one which owns column k, carry out the calculations of local pivoting in parallel, just in case global pivoting is necessary. This prospective calculation

Matrix	*Global pivoting*		*Local pivoting*		%Red.Ex.
	Ex.time	**Error**	**Ex.time**	**Error**	
JPWH991	7.29s	6.88E-15	6.65s	8.22E-15	9%
MAHINDAS	5.66s	6.71E-7	4.43s	2.94E-7	22%
ORSREG1	23.83s	5.23E-13	19.86s	4.73E-13	17%
ORANI678	21.03s	9.92E-13	18.87s	5.62E-13	10%
SHERMAN5	26.95s	4.17E-10	20.88s	1.97E-10	23%
GEMAT11	40.15s	5.03E-5	30.28s	9.81E-5	25%

Table 4.10: *Global pivoting vs local pivoting*

does not increase the computation times.

The drawbacks of local pivoting lie in the fact that, as the candidate pivot column set is restricted, fill-in may increase in some occasions. This is because the criterion to reduce fill-in has less candidate pivot columns to select, among them, the one which causes the maximum preservation of sparsity. This fact could imply higher execution times to treat this new nonzero entries, and this overhead in the execution time can counterbalance the profits in the running times obtained by reducing the communications using local pivoting.

Experimentally, we have proven that the local pivoting strategy provides satisfactory results for high values of *npex* and, in general, for large matrices (n high), in which the effect of the communication reduction is appreciable in the whole factorization execution time. Table 4.10 shows a comparison between global and local pivoting, for the parallel MGS algorithm, on an 8×8 mesh of the Cray T3D, using packed vectors as storage scheme and setting γ=0.99, ϵ=1. Two new sparse matrices from the Harwell-Boeing collection were included because they have adequate dimensions to perceive the profits of local pivoting: *ORSREG1*, a 2205×2205 matrix with 14133 nonzero entries (0.29% of non-null elements) obtained in the field of oil reservoir simulation, and *GEMAT11*, 4929×4929, with 33185 nonzero entries (0.14%), generated in the area of power flow modelling. More experiments on local pivoting using the AP1000 are presented in [95].

As we can see in Table 4.10, execution times are reduced by 17.5% on average using local pivoting, for this set of sparse matrices, which is a significant reduction that results in better efficiencies. Moreover, the numerical errors are practically the same for global and local pivoting, as desired. The numerical error for matrix *GEMAT11* is not very good, but with the Householder and Givens algorithms we obtain, for this matrix, errors of the order of 10^{-10}.

4.5 Sparse QR Factorization on a Vector Processor

As an appendix of this chapter and for illustrative purposes, we run the sparse MGS and Householder algorithms on a vector processor, the Fujitsu VP2400/10

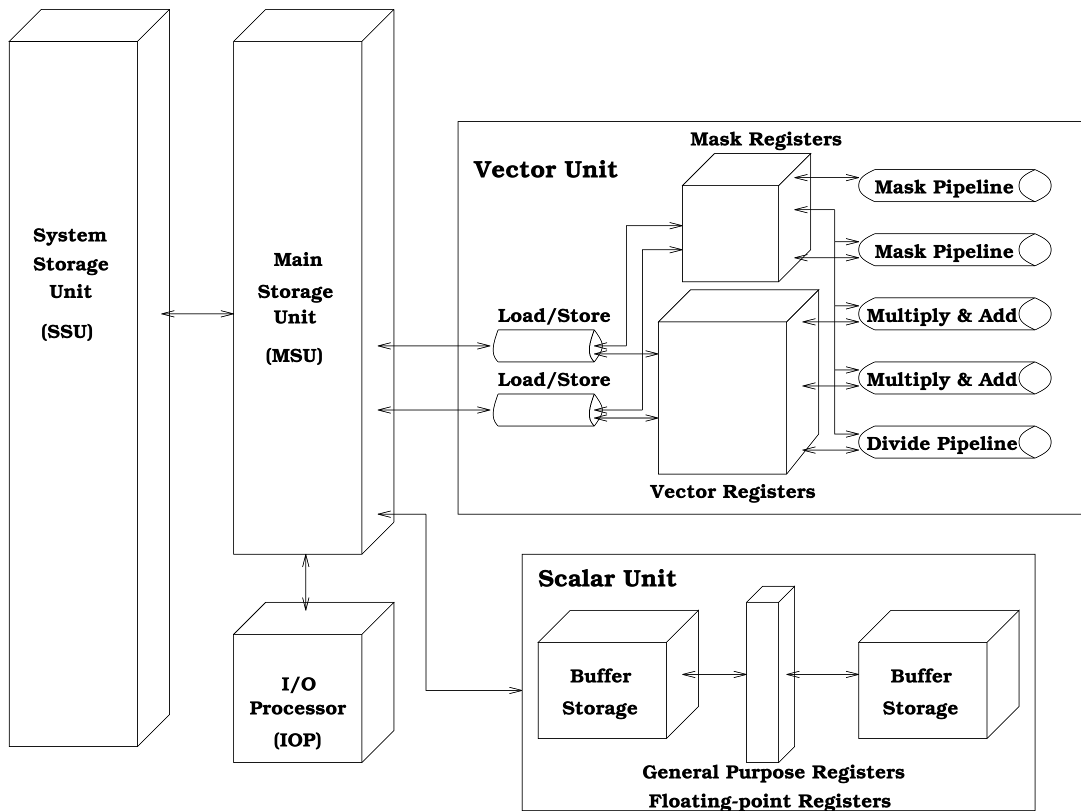

Figure 4.9: *Architecture of the Fujitsu VP2400/10 vector computer*

uniprocessor model, whose structure is shown in Figure 4.9.

It has one scalar and one vector unit. The vector unit of this computer is basically composed of vector registers, mask registers and multiple pipelines. Vector registers are used for storing the data needed by the vector operations and can be reconfigured through hardware in number and word size. Mask registers are used by the conditional statements in order to allow changes in the data flow of the vector operations; this way, conditional statements can be vectorized. The VP-2400/10 model has seven pipelines: two mask pipelines, two for multiplication/addition, two for loading/storage and one division unit. Six pipelines can work concurrently.

The sparse algorithms executed on this computer are the ones which use packed vectors as storage scheme. A set of directives, which begin with the keyword *#pragma*, can be inserted in the code to provide additional information to the vectorizing compiler. Table 4.11 shows the execution times (in seconds) for solving the sparse least squares problem through the MGS and Householder factorizations on the Fujitsu VP2400. Table 2.1 shows the characteristics of the test matrices.

More important than the execution times is the fact that the special characteristics of our sparse codes result in very low vectorization rates, $v = t_{s1}/t_s$ (where t_{s1} is the scalar CPU time of the part of the program which can be vectorized, and t_s is the scalar CPU time for the whole program). Hence, poor performances

Matrix	Ex.time MGS	Ex.time HOU
JPWH991	100.87s	89.18s
SHERMAN1	21.03s	24.33s
MAHINDAS	11.78s	16.22s

Table 4.11: *Execution times on a vector processor*

```
    for (i=0; i<qr.size[j]; i++)
      *(ptr+i) = *(qr.cols[j]+i);

                 (a)

for (j=0; j<n; j++)
  for (ptr=qr.cols[j]; ptr<limit; ptr++)
    buf[ptr->i] += ptr->a;

                 (b)
```

Figure 4.10: *Sparse code examples for a vector processor*

are achieved because the vector unit of Figure 4.9 is under-used. This work is described in [29].

Figure 4.10 shows two significant examples taken from QR algorithms source code (see Section 2.6 and Figure 2.8 for a better comprehension of the code notation). Vectorization is extremely difficult due to the complex data addressing (Figure 4.10a); it even involves a variable of type pointer (*ptr*) in the limits of a loop and its corresponding increment (Figure 4.10b).

As a conclusion, vector computers have been extensively used for years in matrix algebra to deal with large dense matrix problems. However, if matrices are sparse and we use special storage schemes (see the ones of Figure 2.8) for them, vectorization provides a poor performance due to the large number of indirections in the code.

Part II

Compilation Issues of Sparse Factorizations

Chapter 5

A Parallel Library for Sparse Computations

5.1 An Overview of the Library

As we pointed out in the previous chapters, sparse matrix operations appear in many scientific areas. Many libraries have been developed for managing sparse matrices, specially in linear algebra (consult Section 1.4); for instance, SPARSKIT [85], is a set of useful basic routines for sparse matrix manipulation. More libraries for sparse matrix operations are described in [70]. A more recent library is the NIST Sparse BLAS (Basic Linear Algebra Subprogram) library [82], which provides computational kernels for fundamental sparse matrix operations. All of these libraries are based on compressed storage formats which do not consider fill-in.

Many linear algebra applications need to be solved in parallel due to memory and CPU requirements; so, parallel libraries such as *ScaLAPACK* [23] were developed, mainly oriented at dense computations, although they include some sparse matrix features.

In this chapter, we present *3LM*, a C *L*inked *L*ist *M*anagement *L*ibrary which was mainly designed for sparse direct factorizations on distributed memory MIMD computers, in a PVM environment, although these routines can also be applied to other kinds of problems involving fill-in, due to their flexibility. *3LM* is restricted to a mesh topology and is based on an SPMD programming model. This library is based on *DDLY* (Data Distribution Layer) [100], a parallel sparse library suitable for other irregular applications not involving fill-in, such as molecular dynamics and iterative methods for sparse matrices.

Our goal is to make the distributed programming in such environments easier by means of a set of list and vector oriented operations (which can be mixed, if necessary, with message-passing routines for more sophisticated computations). The result is a pseudo-sequential code, in which the IF constructs used to fix the actions on each particular processor are hidden to the programmer. This scheme

also exploits local computations following the owner-computes rule.

5.2 Library Data Structures and Distributions

The choice of a data structure for representing a sparse matrix, as well as the selection of an adequate data distribution are important keys to achieve efficient parallel algorithms. Next, we describe the data structures and distributions available in *3LM*.

5.2.1 Data Schemes

In the context described in the previous section, the scheme we name Linked List Column/Row Distribution (*LLCD/LLRD*) is the one selected for representing and distributing sparse matrices. It includes a data structure: linked lists, each one of them represents one column/row of the matrix, and a data distribution, a pseudo-regular cyclic distribution. There is a detailed description in Sections 2.6 and 3.1, respectively.

For the *LLCD* scheme, this is accomplished by means of the following *3LM* routine:

```
int  lll_dcd (char *file_n, int n, dll *list_id)
```

where *file_n* is the name of the file in which the matrix is stored in Harwell-Boeing format [34] or in coordinate format; if *file_n*=0 (or NULL), the structure is only set up (this is for sparse matrices which are generated at run-time); n is the number of columns of the matrix, and *list_id* is an identifier (or descriptor) of the matrix which contains the list orientation, and an array of pointers to the beginning (*first*) and to the end (*last*) of the lists. The type of this identifier is the predefined data type *dll*.

In Figure 5.1 an 8×8 matrix is distributed onto a 2×2 mesh using this scheme. Similar routines have been developed for a row-oriented scheme: *lll_drd* (in this case, each list represents one row of the sparse matrix), for singly-linked lists: *lll_scd, lll_srd* (column and row-oriented, respectively), and even for two-dimensional linked lists: *lll_srcd, lll_drcd* (singly and doubly-linked, in this order). In these cases the library predefined types of the identifiers are different for each scheme (for instance, *sll* for one-dimensional singly-linked lists).

In matrix algebra, vector operations (scalar-vector multiplication, vector addition, dot product, vector multiply, saxpy operation) are very common. Besides, the characteristics of many problems force the programmer to manage vectors distributed (and even replicated) in a row or column of processors of the virtual mesh to take advantage of data locality and to minimize communications. In order to make this kind of operations easy, we use the routine:

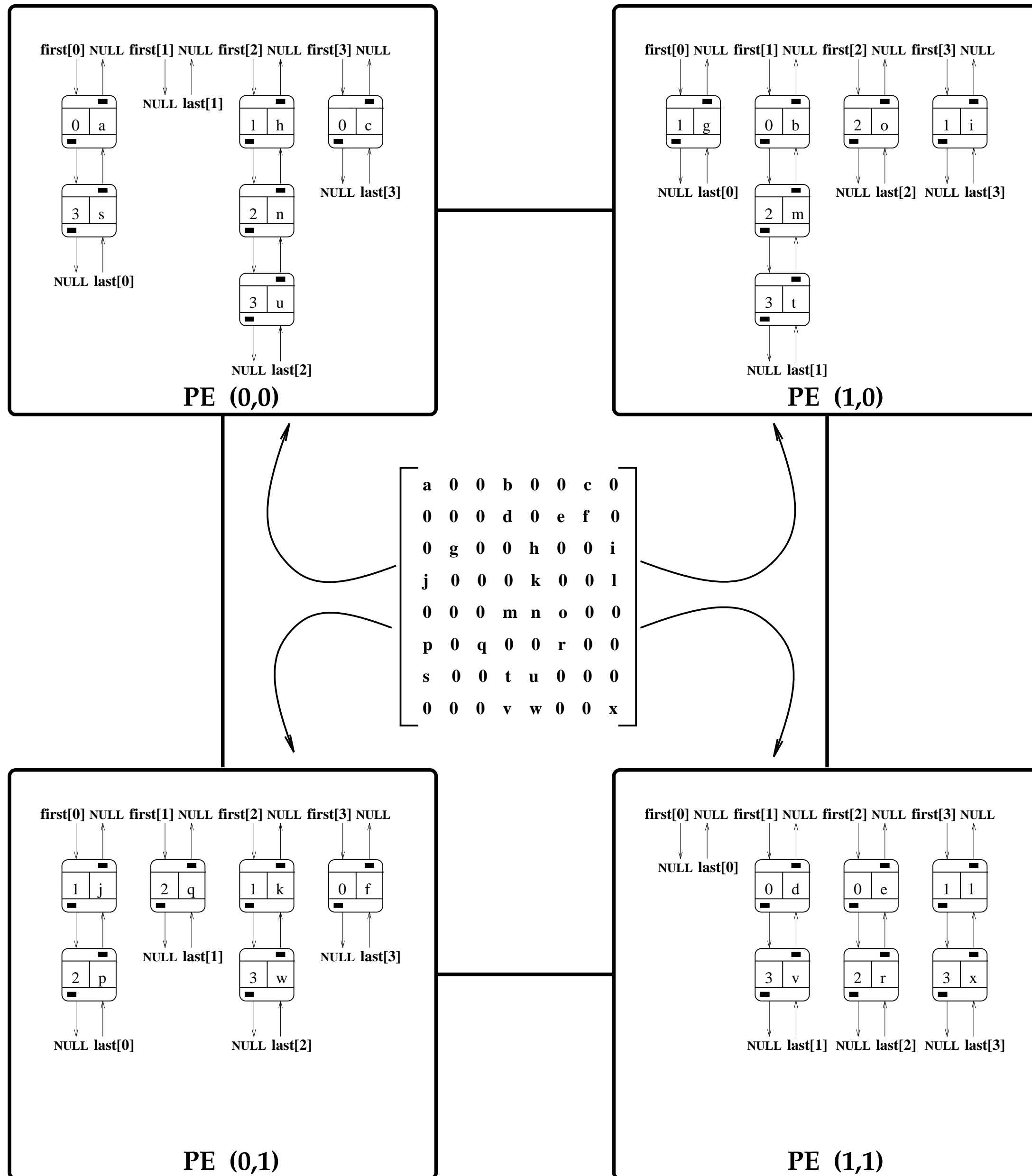

Figure 5.1: *LLCD scheme*

```
void *lll_InitVector (char *file_n, int nels, vector *vec_id,
                      int dir, int dtype)
```

which distributes a dense vector of *nels* components stored in *file_n* onto a mesh in a cyclic way, over each row or column of processors (that is, replicated), depending on *dir* (*XDirection* or *YDirection*); *dtype* is the data type of the elements: *DataInt, DataFloat, DataDouble* (predefined constants). This routine stores in *vec_id* (vector identifier) the following information: orientation of the vector (row-oriented or column-oriented), type of the elements and a pointer to the beginning of the vector, which is also returned.

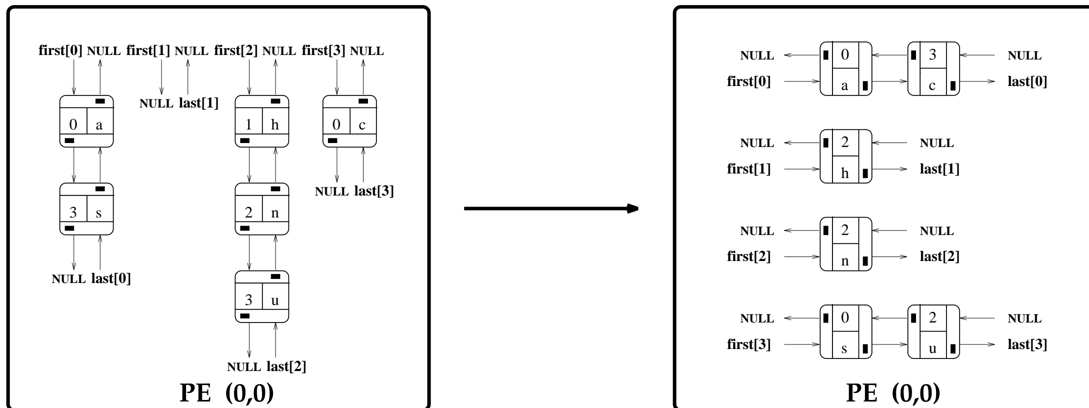

Figure 5.2: *List reorientation, from column-direction to row-direction*

5.2.2 Data Reorientation

The *LLCD* and *LLRD* schemes favour a column and a row access to the elements of the matrix, respectively. In many matrix problems, the predominant direction of the accesses to the matrix entries may vary during program execution. Therefore, it would be interesting to change the orientation of the data storage scheme at run-time in a point of the code to allow efficient data access in other direction. The penalization of this reorientation can compensate for the overhead of performing, for instance, many row accesses to data stored in a column-oriented fashion.

The procedure *lll_dreorient* offers this chance:

```
lll_dreorient(&list_id, new_direction, m, n)
```

where $m \times n$ are the dimensions of the matrix, defined by *list_id*, to be reoriented; *new_direction* is the new orientation of the matrix and it can be *XDirection* or *YDirection*.

This operation does not imply communications (it is a local operation) or allocations/releases of memory; it only involves a pointer reassignment, as we can see in Figure 5.2, where the column-oriented (*YDirection*) data storage scheme of Figure 5.1 is transformed into a row-oriented (*XDirection*) one (this is only shown for data of PE (0,0)). The routine *lll_dreorient* is for one-dimensional doubly-linked lists, but there is also an analogous routine for singly-linked lists (*lll_sreorient*). Obviously, there are no reorientation routines for two-dimensional linked lists, as they allow efficient access both by rows and columns.

Similarly, we have the routine *lll_vreorient* to redistribute vectors. For instance, if we want to reorient a column-oriented vector (previously initialized by routine *lll_InitVector* and identified by *v_id*) to a row-oriented vector, we should write:

```
v=lll_vreorient(&v_id, XDirection, nels)
```

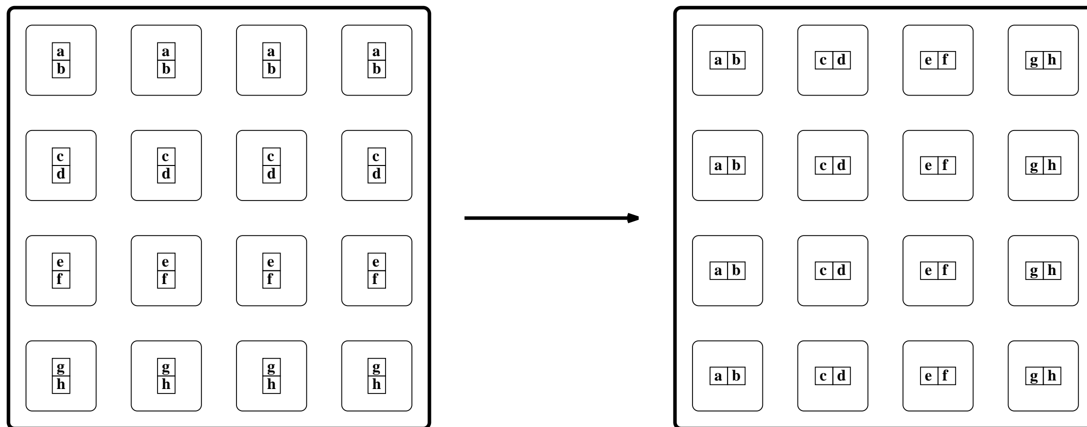

Figure 5.3: *Vector redistribution, from column-direction to row-direction*

where the first parameter is the identifier of the vector, the second one is the new direction of the vector and the last one is its size. It returns a pointer to the beginning of the vector (v) in order to work with it using the vector notation. It is recommended to use the same name for the vector (v in the example) as the old one; otherwise we must take into account than we cannot refer to the new vector with the old name, which has no associated vector any more. This is because this procedure (unlike *lll_dreorient*) implies allocations/releases of memory. It also entails communications to redistribute the entries of the vector in the corresponding processors. Figure 5.3 is an example of this kind of redistribution. If we reorient a vector in the same direction it previously had,

```
new_v=lll_vreorient(&v_id, YDirection, nels)
```

the effect is that we can refer to this vector using the old name (v in the example) or the new one *new_v*, because they are physically the same vector.

5.3 Mapping Iterations

Let us assume a mesh topology in which each processor is identified by coordinates *(pidx, pidy)*, being *pidx* the coordinate in the X axis of the mesh and *pidy* the coordinate in the Y axis. All programs must begin with the statement:

```
lll_BeginMesh (npex, npey, progname)
```

which sets a mesh of dimensions $npey \times npex$ and executes the program *progname* in all the processors, using a PVM environment, for a cluster of workstations or for a Cray T3D/T3E supercomputer (by setting *progname*=0/NULL). Programs must end with *lll_EndMesh()*.

As an example, considering that a matrix A is stored in an $LLCD$ format, a double loop which performs a column access is mapped on a local double loop as shown in Figure 5.4.

```
for (j=j1; j<j2; j++)                for (j=f_col(j1); j<f_col(j2); j++)
    for (i=i1; i<i2; i++)                for (i=f_row(i1); i<f_row(i2); i++)
        A_ij = ···                           A_ij = ···
```

being

$$f_{col}(x) = \left\lfloor \frac{x}{npex} \right\rfloor + \begin{cases} 1 & \text{if } pidx < (x \; mod \; npex) \\ 0 & \text{otherwise} \end{cases}$$

$$f_{row}(x) = \left\lfloor \frac{x}{npey} \right\rfloor + \begin{cases} 1 & \text{if } pidy < (x \; mod \; npey) \\ 0 & \text{otherwise} \end{cases}$$

Figure 5.4: *Mapping global loops onto local loops*

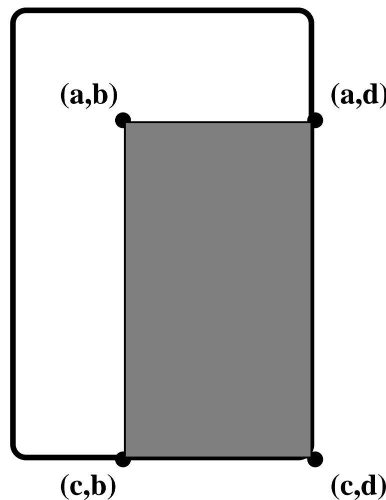

Figure 5.5: *An example of iteration space*

Let us consider the iteration space of Figure 5.5. It is very common in problems that involve the reduction of the index space, such as the LU, QR and Cholesky factorizations, in which submatrix operations are very usual. For instance, let us consider the operation of multiplying the entries of submatrix $A_{a:c,b:d}$, shown in Figure 5.5 by a constant named *value*. The corresponding SPMD code using *3LM* for the processor mesh is expressed as:

```
for (j=fcol(b); j<fcol(d+1); j++)
   lll_doper(j, listA_id, value, OpMul, frow(a), frow(c+1));
```

where *listA_id* is the matrix A identifier, *OpMul* is the operation (product) of each element of the list with *value*, from global row index a up to index c. There are predefined operations: *OpAdd, OpSub, OpMul, OpDiv, Nop* (this last one defined

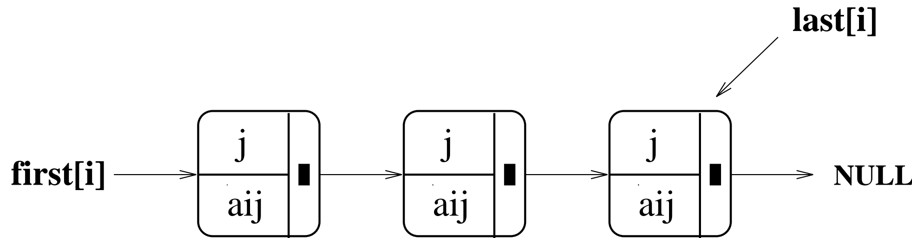

Figure 5.6: *Singly-linked list data structure*

as *Nop(a,b)=b*), as well as user-defined ones (consult Subsection 5.6.3). As we can see, all the iterations of the outer loop are traversed, but the inner loop is converted into the procedure *lll_doper* in order to go only through the links of the corresponding list because it is not necessary to traverse all the iterations.

An analogous procedure for vectors is *lll_doperv*, with the same parameters as *lll_doper*, but with a vector identifier *vec_id* instead of a constant *value*; it operates each element of list *j* with the corresponding entry of *vec_id*.

5.4 Library Routines

The *3LM* routines we have shown in the previous sections and the ones we will see next have been specified for column-oriented operations, that is, using an *LLCD* scheme for lists and using column vectors (*YDirection*). This was accomplished in order to simplify their explanation. There exist similar procedures for singly-linked lists (*lll_s**) and for 2-D linked lists (*lll_2d**, *lll_2s**).

However, the same routines can be also applied to row-oriented operations (when using an *LLRD* scheme and row vectors) because these routines obtain the orientation from the identifiers of lists and vectors, and they operate accordingly. Therefore, in the sections of this chapter in which the term *'column j'* appears, it can be also applied to *'row i'*, depending on the distribution scheme used. For instance, a one-dimensional row-oriented singly-linked list is shown in Figure 5.6. Note that, as in doubly-linked lists, we have a pointer to the end of each list; this is to make insertion and deletion operations easier.

5.4.1 Replication Operations

Sometimes, a column (in an *LLCD* scheme) of the matrix is required to perform calculations with data located in other processors. The procedure:

```
lll_drepl(j, list_id, vec_id, low, high)
```

replicates column *j* of the matrix on the corresponding processors, in the dense vector defined by *vec_id*, from entry with index *low* up to entry *high*, not inclusive

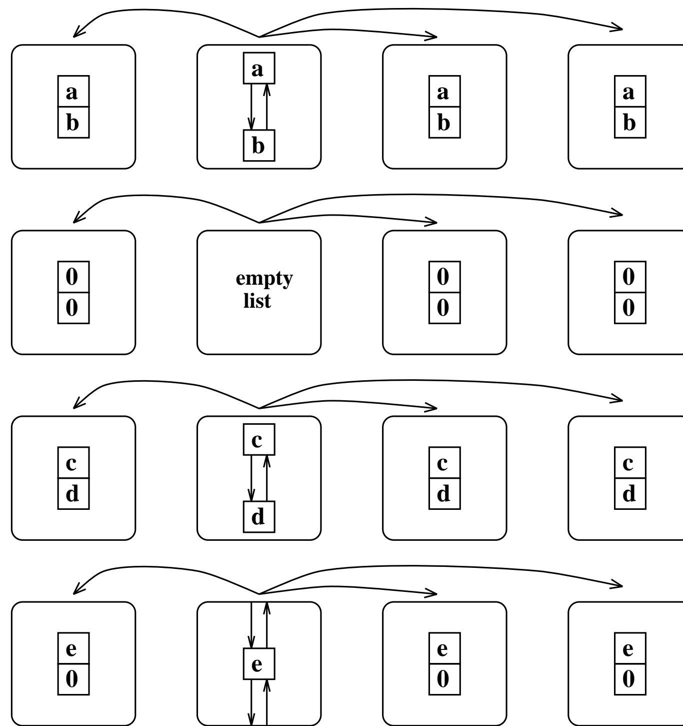

Figure 5.7: *Replication of one column of a matrix*

(this fact makes it applicable to all functions in which parameters *low* and *high* appear).

Regarding implementation details, this procedure broadcasts a compressed vector instead of a full-size vector to reduce the size of the message to be broadcast. Figure 5.7 shows an example of this operation. There are analogous procedures to replicate dense vectors (*lll_vrepl*).

5.4.2 Gather Operations

They are used for vectors which are replicated (but not necessarily) and cyclically distributed on each row (or column) of processors, and other processors need to obtain non-local data of these vectors. The function:

```
lll_vgather(vec_id, j1, j2)
```

returns the value of entry *j1* of the vector identified by *vec_id* to the processors which own entry *j2*. If *j2*=*All* (*All* is a predefined constant of the library), this value is returned to all the processors, as shown in Figure 5.8.

5.4.3 Reduction Routines

3LM provides a set of reduction instructions, both for lists and for dense vectors. For instance:

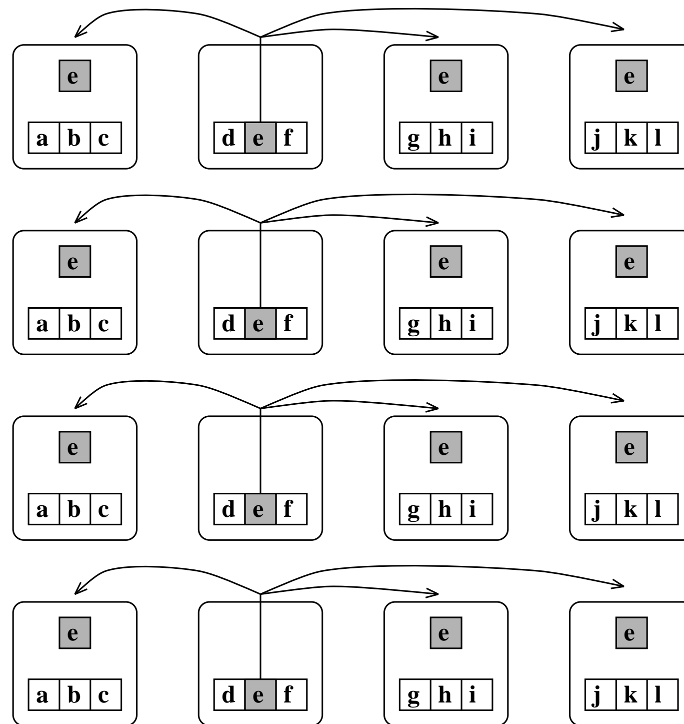

Figure 5.8: *Gather operation*

```
lll_vmaxval/lll_vminval(vec_id, low, high)
```

returns the maximum/minimum element of vector *vec_id*, from index *low* up to index *high*. Similarly,

```
lll_vmaxloc/lll_vminloc(vec_id, low, high)
```

returns the index of the maximum/minimum element. There are also reduction routines for arithmetic operations (sum, product, etc.) and even for user-defined operations (see Subsection 5.6.3).

5.4.4 Fill-in Routines

In the sparse computations we are considering, an important drawback is the generation of new nonzero entries in the matrix, with the corresponding problems of storage and treatment of these entries. This is solved by means of the linked list structure. Let us consider the following double loop which traverses the iteration space of Figure 5.5:

for (j=b; j<d+1; j++)
    for (i=a; i<c+1; i++)
        $A_{ij} = A_{ij} + vec_i$

where *vec* is a vector previously distributed and replicated in each column of processors. Fill-in appears in this computation and is confined in the local processor

which executes its own set of iterations. We can solve this using the following routine:

```
for (j=fcol(b); j<fcol(d+1); j++)
   lll_dfillin(j, listA_id, vec_id, OpAdd, frow(a), frow(c+1));
```

where *OpAdd* is the operation between the elements of the list and the elements of vector *vec*. Generalizing, this operation can be a predefined (see Section 5.3) or even a user-defined function g_{oper}. According to this, the procedure *lll_dfillin* carries out the following actions:

$$If\ A^*_{ij} \leftarrow g_{oper}(A_{ij}, vec_i) \begin{cases} \neq 0\ and\ A_{ij} \neq 0 & Entry\ A_{ij}\ updated\ in\ the\ list\ as\ A^*_{ij} \\ \neq 0\ and\ A_{ij} = 0 & New\ entry\ A^*_{ij}\ inserted\ in\ the\ list \\ 0 & Entry\ A_{ij}\ deleted\ of\ the\ list \\ A_{ij} & No\ actions\ are\ taken \end{cases}$$

For this routine, g_{oper} can be the predefined function *Nop*, being, for this particular example, $Nop(A_{ij},vec_i)=vec_i$. So, function *Nop*, as parameter of the procedure *lll_dfillin* explained above, is equivalent to delete list j, and the new list (column) j corresponds now to the nonzero entries of vector *vec*.

A complete set of insertion and deletion functions, hidden to the programmer, has been developed to manage the fill-in in the list structures. There also exists a constant value named *accuracy*, which can be set in the code; it is a threshold so that all matrix entries with absolute value less than *accuracy* are considered as zeroes (by default, for double precision numbers, *accuracy*$=10^{-20}$). This constant has the same purpose as the threshold value ϕ of expressions (2.3) and (2.18).

A routine which is related to *lll_dfillin* is:

```
lll_dupdate(i, j, aij, list_id)
```

which sets element *(i,j)* in the matrix identified by *list_id* to *aij* (as in *lll_dfillin* insertion and deletion are assumed depending on the value of *aij*). In an *LLCD* scheme, this procedure can be useful for updating a row of the matrix. Note that, for this scheme, a row access is very expensive in comparison with a column access. For instance, if we want to set the elements of row $A_{a+3,b:d-3}$ (see Figure 5.5) to the corresponding values of a row vector named *new_values*, we should write:

```
for (j=fcol(b); j<fcol(d-2); j++)
   for (i=frow(a+3); i<frow(a+4); i++)
      lll_dupdate(i, j, new_values[j], listA_id);
```

The inner loop is necessary to set the PEs which do not own row *a+3* to idle.

5.4.5 Swapping Operation

In many matrix calculations, explicit pivoting operations are required. This feature is a drawback in sparse computations due to the storage scheme and to

the fill-in, which changes the number of elements per column of the matrix. A high-level swap operation is implemented to make the programming easier:

```
lll_dswap(j1, j2, list_id, m)
```

being *j1* and *j2* the global indices of the columns to be swapped in the mesh according to the *LLCD* scheme, and m is the row dimension of the matrix. Compressed vectors are used to reduce message sizes.

5.4.6 Other Routines

Additional remarkable procedures for list management are:

- `lll_dcount(j, list_id, low, high)`

  returns the number of elements of list (column) j defined by *list_id*, from row index *low* up to index *high* (not inclusive). This value is returned to all the processors which contain column j.

- `lll_dvdp(j, list_id, vec_id, low, high)`

  returns to all the processors which own list (column) j the dot product between that column and vector *vec_id*, with *low* and *high* as index limits. There is a similar function *lll_vvdp* which obtains the dot product of two vectors.

- `lll_dunpack(j, list_id, vec_id, low, high)`

  copies elements of list (column) j, from index *low* up to index *high* on the corresponding positions of the column vector defined by *vec_id*; the rest of elements of this vector are zero.

- `lll_dpop(i, j, list_id)`

  obtains entry *(i,j)* of the list defined by *list_id*. If that element is not in the list, it returns zero.

There are also *3LM* low-level routines to manage the data structures directly, as well as to determine the actions on each processor of the mesh, for special operations which cannot be performed with the high-level set described above. These routines are described in Section 5.6.

5.5 Practical Examples

The following subsections present examples of parallel sparse algorithms programmed with the *3LM* routines, focusing on sparse QR codes and least squares problems.

```
#include "lll.h"

void main()
{
 int m, n, pesx, pesy, j;
 double *temp;
 vector temp_id;
 sll listA_id, listB_id;

 pesx=4; pesy=4;
 lll_BeginMesh(pesx, pesy, "addAB");
 m=5000; n=5000;
 temp=lll_InitVector(0, m, &temp_id, YDirection, DataDouble);
 lll_scd("matrixA", n, &listA_id);
 lll_scd("matrixB", n, &listB_id);
 for (j=fcol(0); j<fcol(n); j++) {
    lll_sunpack(j, listB_id, temp_id, frow(0), frow(m));
    lll_sfillin(j, listA_id, temp_id, OpAdd, frow(0), frow(m));
   }
 lll_EndMesh();
}
```

Figure 5.9: *Column-oriented sparse matrix-sparse matrix sum*

```
#include "lll.h"

void main()
{
 int m, n, pesx, pesy, i;
 double *temp;
 vector temp_id;
 sll listA_id, listB_id;

 pesx=4; pesy=4;
 lll_BeginMesh(pesx, pesy, "addAB2");
 m=5000; n=5000;
 temp=lll_InitVector(0, n, &temp_id, XDirection, DataDouble);
 lll_srd("matrixA", m, &listA_id);
 lll_srd("matrixB", m, &listB_id);
 for (i=frow(0); i<frow(m); i++) {
    lll_sunpack(i, listB_id, temp_id, fcol(0), fcol(n));
    lll_sfillin(i, listA_id, temp_id, OpAdd, fcol(0), fcol(n));
   }
 lll_EndMesh();
}
```

Figure 5.10: *Row-oriented sparse matrix-sparse matrix sum*

5.5.1 Sparse Matrix-Sparse Matrix Sum

As a first example of the use of these routines, we display in Figure 5.9 the code for an in-place sparse matrix-sparse matrix sum (an operation with fill-in), $A \leftarrow A + B$, on a 4×4 mesh, using singly-linked lists. Matrices A and B are read from files and their dimensions are 5000×5000. A row-oriented version is shown in Figure 5.10.

```
#include "lll.h"

void main()
{
 int m, n, pesx, pesy;
 int i, j, k, rank, p;
 double pivot, tempnorm;
 double *norm, *vsum, *vcol, *temp;
 vector norm_id, vsum_id, vcol_id, temp_id;
 dll listQ_id, listR_id;

 pesx=4; pesy=4;
 lll_BeginMesh(pesx, pesy, "qr_mgs");
 m=100; n=100;
 norm=lll_InitVector(0, n, &norm_id, XDirection, DataDouble);
 vsum=lll_InitVector(0, n, &vsum_id, XDirection, DataDouble);
 vcol=lll_InitVector(0, m, &vcol_id, YDirection, DataDouble);
 temp=lll_InitVector(0, m, &temp_id, YDirection, DataDouble);
 lll_dcd("matrix.dat", n, &listQ_id);
 lll_dcd(NULL, n, &listR_id);
 rank=n;
 for (j=fcol(0); j<fcol(n); j++) {
    lll_dunpack(j, listQ_id, temp_id, frow(0), frow(m));
    norm[j]=lll_dvdp(j, listQ_id, temp_id, frow(0), frow(m));
   }
 for (k=0; k<n; k++) {
    pivot=lll_vmaxval(norm_id, fcol(k), fcol(n));
    p=lll_vmaxloc(norm_id, fcol(k), fcol(n));
    if (pivot < 1.0e-20) {
       rank=k; break;
      }
    lll_dswap(k, p, listQ_id, m);
    lll_dswap(k, p, listR_id, n);
    tempnorm=lll_vgather(norm_id, k, p);
    for (j=fcol(p); j<fcol(p+1); j++)
       norm[j]=tempnorm;
    pivot=sqrt(pivot);
    for (j=fcol(k); j<fcol(k+1); j++) {
       for (i=frow(k); i<frow(k+1); i++)
          lll_dupdate(i, j, pivot, listR_id);
       lll_doper(j, listQ_id, pivot, OpDiv, frow(0), frow(m));
      }
    lll_drepl(k, listQ_id, vcol_id, 0, m);
    for (j=fcol(k+1); j<fcol(n); j++) {
       vsum[j]=lll_dvdp(j, listQ_id, vcol_id, frow(0), frow(m));
       for (i=frow(k); i<frow(k+1); i++)
          lll_dupdate(i, j, vsum[j], listR_id);
       norm[j]=norm[j]-vsum[j]*vsum[j];
       for (i=frow(0); i<frow(m); i++)
          temp[i]=vcol[i]*vsum[j];
       lll_dfillin(j, listQ_id, temp_id, OpSub, frow(0), frow(m));
      }
   }
 lll_EndMesh();
}
```

Figure 5.11: *Sparse MGS code using 3LM routines*

5.5.2 Sparse QR Factorizations

Sparse QR algorithms are more complex examples of programming with this library. The code of Figure 5.11 shows an example of the use of the *3LM* routines

```
for (j=fcol(k); j<fcol(n); j++) {
   if (norm[j] < 1.0e-20)
      counter[j]=m+1;
   else counter[j]=lll_dcount(j, listQ_id, frow(0), frow(m));
  }
p=lll_vminloc(counter_id, fcol(k), fcol(n));
pivot=lll_vgather(norm_id, p, All);
```

Figure 5.12: *Code to preserve sparsity in the MGS algorithm*

for the rank-revealing sparse MGS algorithm, with column pivoting, whose manual parallel implementation was widely described in Section 3.2. As we can see, this code is not very broad (as much as the sequential code, more or less), whereas the corresponding parallel code written by hand fills two thousand lines of code, approximately.

The pivoting strategy applied in the example of Figure 5.11 is the one used in dense QR factorizations (see expression (2.2)). Nevertheless, it is easy to program, using this library, the fill-in strategy explained in Subsection 2.7.1. This is accomplished by substituting the first two lines of code of the main loop k in Figure 5.11 by the code of Figure 5.12. The integer vector *counter* which appears in this figure was previously declared and set as a row vector:

```
counter=lll_InitVector(0, n, &counter_id, XDirection, DataInt)
```

This *3LM* code uses expression (2.18) with $\epsilon = 1$, that is, it selects as pivot column the one with the least number of nonzero elements, provided that the square of the norm of the pivot column is larger than a specific threshold (10^{-20} in our example code).

Figure 5.13 is the *3LM* code for the sparse Householder transformations (see Section 3.3 for more details about the parallel algorithm), with the fill-in reduction method incorporated.

Note that, in these codes, local indices are always used, both in the loop limits and in the routines of the library. It is achieved by means of functions *fcol* and *frow*, previously described in Figure 5.4. This allows the appropriate processors to be working. The exceptions are the routines for replication (Subsection 5.4.1), gathering (Subsection 5.4.2) and swapping (Subsection 5.4.5), which use global indices in their arguments. The set of reduction instructions *maxloc/minloc* (see Subsection 5.4.3) also returns a global index.

As we said before, in this chapter we have mainly focused on column-oriented operations, but these routines are also applied to row-oriented operations depending on the distribution of lists and vectors. For instance, in the code of Figure 5.13, the first routine *lll_vrepl* is applied to a column vector *v_id* (replication in each column of processors), but the second one is applied to a row vector *temp2_id* (replication in each row of processors). In order to show this feature,

```
#include "lll.h"
void main()
{
 int m, n, pesx, pesy, i, j, k, rank, p;
 double pivot, tempnorm, divider, beta;
 double *norm, *w, *v, *temp, *temp2;
 int *counter;
 vector norm_id, w_id, v_id, temp_id, temp2_id, counter_id;
 dll listQR_id;
 pesx=4; pesy=4; m=100; n=100;
 lll_BeginMesh(pesx, pesy, "qr_hou");
 norm=   lll_InitVector(0, n, &norm_id,    XDirection, DataDouble);
 w=      lll_InitVector(0, n, &w_id,       XDirection, DataDouble);
 v=      lll_InitVector(0, m, &v_id,       YDirection, DataDouble);
 temp=   lll_InitVector(0, m, &temp_id,    YDirection, DataDouble);
 temp2=  lll_InitVector(0, n, &temp2_id,   XDirection, DataDouble);
 counter=lll_InitVector(0, n, &counter_id, XDirection, DataInt);
 lll_dcd("matrix.dat", n, &listQR_id);
 rank=n;
 for (j=fcol(0); j<fcol(n); j++) {
    lll_dunpack(j, listQR_id, temp_id, frow(0), frow(m));
    norm[j]=lll_dvdp(j, listQR_id, temp_id, frow(0), frow(m));
   }
 for (k=0; k<n; k++) {
    for (j=fcol(k); j<fcol(n);j++) {
       if (norm[j] < 1.0e-20)
          counter[j]=m-k+1;
       else counter[j]=lll_dcount(j, listQR_id, frow(k), frow(m));
      }
    p=lll_vminloc(counter_id, fcol(k), fcol(n));
    pivot=lll_vgather(norm_id, p, All);
    if (pivot < 1.0e-20) {
       rank=k; break;
      }
    lll_dswap(k, p, listQR_id, m);
    tempnorm=lll_vgather(norm_id, k, p);
    for (j=fcol(p); j<fcol(p+1); j++)
       norm[j]=tempnorm;
    pivot=sqrt(pivot);
    for (j=fcol(k); j<fcol(k+1); j++) {
       lll_dunpack(j, listQR_id, v_id, frow(k), frow(m));
       divider=lll_vgather(v_id, k, All);
       if (divider < 0.0)
          pivot=-pivot;
       divider=divider+pivot;
       for (i=frow(k); i<frow(m); i++)
          v[i]=v[i]/divider;
      }
    lll_vrepl(k, v_id, k, m);
    for (i=frow(k); i<frow(k+1); i++)
       v[i]=1.0;
    beta=-2/lll_vvdp(v_id, v_id, frow(k), frow(m));
    for (j=fcol(k); j<fcol(n); j++) {
       w[j]=beta*lll_dvdp(j, listQR_id, v_id, frow(k), frow(m));
       for (i=frow(k); i<frow(m); i++)
          temp[i]=v[i]*w[j];
       lll_dfillin(j, listQR_id, temp_id, OpAdd, frow(k), frow(m));
       temp2[j]=lll_dpop(frow(k), j, listQR_id);
      }
    lll_vrepl(k, temp2_id, k+1, n);
    for (j=fcol(k+1); j<fcol(n); j++)
       norm[j]=norm[j]-temp2[j]*temp2[j];
   }
 lll_EndMesh();
}
```

Figure 5.13: *Sparse Householder code using 3LM routines*

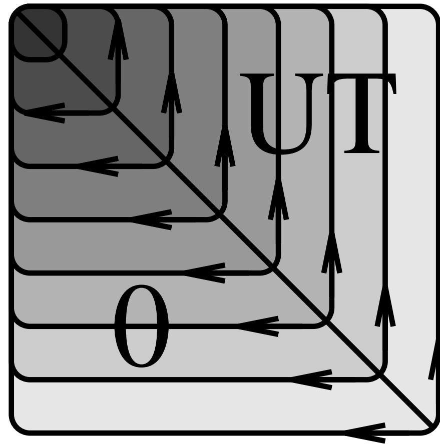

Figure 5.14: *Row-oriented Householder transformations*

we have implemented a row-oriented triangularization of a sparse square matrix by means of Householder transformations using *3LM* routines. The procedure is based on Householder post-multiplications [42, Chapter 5]. Row pivoting was also incorporated to preserve sparsity. This algorithm traverses the iteration space in descending order, and the subrow *k,0:k-1* is zeroed in the *k-th* iteration; this way, the lower triangular part of the matrix is annihilated and the upper triangular matrix *UT* is obtained, as shown in Figure 5.14. The corresponding *3LM* code is presented in Figure 5.15 (note that this algorithm is not exactly a QR factorization, it is only a matrix upper triangularization to show the handling of the *3LM* routines).

5.5.3 Sparse Least Squares Problems

We have also implemented the least squares problem by means of the sparse MGS code of Figure 5.11. As we detailed in Section 3.5, this problem is approached by solving the upper triangular system $R\Pi^T x = Q^T b$, which consists of three stages: calculation of $Q^T b$, back-substitution and permutation.

First, the following vectors must be declared and initiated in the *3LM* code:

- Vector b (right-hand side of the system)

```
double *b;
vector b_id;
b=lll_InitVector("vectorb.dat", m, &b_id, YDirection,
                 DataDouble);
```

- Vector *qtb* (stores the product $Q^T b$)

```
double *qtb;
vector qtb_id;
qtb=lll_InitVector(0, n, &qtb_id, XDirection, DataDouble);
```

```
#include "lll.h"
void main()
{
 int dim, pesx, pesy, i, j, k, p;
 double pivot, tempnorm, divider, beta;
 double *norm, *w, *v, *temp, *temp2;
 int *counter;
 vector norm_id, w_id, v_id, temp_id, temp2_id, counter_id;
 dll listUT_id;
 pesx=4; pesy=4; dim=100;
 lll_BeginMesh(pesx, pesy, "triang_hou");
 norm=   lll_InitVector(0, dim, &norm_id,    YDirection, DataDouble);
 w=      lll_InitVector(0, dim, &w_id,       YDirection, DataDouble);
 v=      lll_InitVector(0, dim, &v_id,       XDirection, DataDouble);
 temp=   lll_InitVector(0, dim, &temp_id,    XDirection, DataDouble);
 temp2=  lll_InitVector(0, dim, &temp2_id,   YDirection, DataDouble);
 counter=lll_InitVector(0, dim, &counter_id, YDirection, DataInt);
 lll_drd("matrix.dat", dim, &listUT_id);
 for (i=frow(0); i<frow(dim); i++) {
    lll_dunpack(i, listUT_id, temp_id, fcol(0), fcol(dim));
    norm[i]=lll_dvdp(i, listUT_id, temp_id, fcol(0), fcol(dim));
   }
 for (k=dim-1; k>-1; k--) {
    for (i=frow(0); i<frow(k+1);i++) {
       if (norm[i] < 1.0e-20)
          counter[i]=k+1;
       else counter[i]=lll_dcount(i, listUT_id, fcol(0), fcol(k+1));
      }
    p=lll_vminloc(counter_id, frow(0), frow(k+1));
    pivot=lll_vgather(norm_id, p, All);
    lll_dswap(k, p, listUT_id, dim);
    tempnorm=lll_vgather(norm_id, k, p);
    for (i=frow(p); i<frow(p+1); i++)
       norm[i]=tempnorm;
    pivot=sqrt(pivot);
    for (i=frow(k); i<frow(k+1); i++) {
       lll_dunpack(i, listUT_id, v_id, fcol(0), fcol(k+1));
       divider=lll_vgather(v_id, k, All);
       if (divider < 0.0)
          pivot=-pivot;
       divider=divider+pivot;
       for (j=fcol(0); j<fcol(k+1); j++)
          v[j]=v[j]/divider;
      }
    lll_vrepl(k, v_id, 0, k+1);
    for (j=fcol(k); j<fcol(k+1); j++)
       v[j]=1.0;
    beta=-2/lll_vvdp(v_id, v_id, fcol(0), fcol(k+1));
    for (i=frow(0); i<frow(k+1); i++) {
       w[i]=beta*lll_dvdp(i, listUT_id, v_id, fcol(0), fcol(k+1));
       for (j=fcol(0); j<fcol(k+1); j++)
          temp[j]=v[j]*w[i];
       lll_dfillin(i, listUT_id, temp_id, OpAdd, fcol(0), fcol(k+1));
       temp2[i]=lll_dpop(i, fcol(k), listUT_id);
      }
    lll_vrepl(k, temp2_id, 0, k+1);
    for (i=frow(0); i<frow(k+1); i++)
       norm[i]=norm[i]-temp2[i]*temp2[i];
   }
 lll_EndMesh();
}
```

Figure 5.15: *Row-oriented sparse upper triangularization using Householder transformations (3LM code)*

```
for (k=rank-1; k>-1; k--) {
   for (i=frow(k); i<frow(k+1); i++) {
      rbyx=lll_dvdp(i, listR_id, x_id, fcol(k+1), fcol(rank));
      for (j=fcol(k); j<fcol(k+1); j++)
         x[j]=(qtb[j]-rbyx)/lll_dpop(i, j, listR_id);
     }
   lll_vrepl(k, x_id, k, k+1);
  }
```

Figure 5.16: *3LM back-substitution*

- Vector x (solution vector)

  ```
  double *x;
  vector x_id;
  x=lll_InitVector(0, n, &x_id, XDirection, DataDouble);
  ```

- An integer vector *permute* $\in \Re^n$. As the entries of this vector are not distributed over the processors, it does not require the *lll_InitVector 3LM* routine. Entry k of this vector contains the index of the pivot column (p) in iteration k. This is achieved by adding the following statement at the end of the code of Figure 5.12:

  ```
  permute[k]=p;
  ```

The product Q^Tb is obtained in vector *qtb* by means of the following *3LM* sentence, located just after the routine *lll_doper* (inside the loop j) of the code of Figure 5.11:

```
qtb[j]=lll_dvdp(j, listQ_id, b_id, frow(0), frow(m));
```

Once we have calculated Q^Tb and the factorization ends, a back-substitution (3.2) is required to solve the system $Rx = Q^Tb$. As we pointed out in Section 3.5, one possible approach is to change the storage scheme of matrix R to a row-oriented one; this solution is the one chosen because it is easy to implement by means of the *3LM* reorientation routines (see Subsection 5.2.2), as opposed to the auxiliary pointer vector approach chosen in the manual codes.

Therefore, matrix R is previously reoriented to a row-oriented scheme using this sentence:

```
lll_dreorient(&listR_id, XDirection, n, n);
```

and then, the corresponding row-oriented back-substitution is applied, as displayed in Figure 5.16.

But the least squares procedure is not finished yet. The permutations performed in the algorithm must be applied to the elements of the solution vector x. Therefore, we must apply the swaps stored in vector *permute*, starting from the end, to obtain the elements of vector x in the right order (see code of Figure 5.17).

```
for (k=rank-1; k>-1; k--) {
   buf1=lll_vgather(x_id, k, permute[k]);
   buf2=lll_vgather(x_id, k, permute[k], k);
   for (j=fcol(k); j<fcol(k+1); j++)
      x[j]=buf2;
   for (j=fcol(permute[k]); j<fcol(permute[k+1]); j++)
      x[j]=buf1;
  }
```

Figure 5.17: *Permutation of the entries of the solution vector x*

5.6 Low-Level Features

In special and unavoidable situations, users need to program in a low-level fashion, taking into account the mesh topology and using message-passing routines directly. The functions of the next subsections (none of these functions were necessary in the example codes of the previous sections) are provided to aid the programmer in these special tasks.

5.6.1 Mesh Functions

The mesh topology is visible for the programmer through the use of the following functions:

- `npes()`. Returns the number of processors of the mesh.
- `npesx/npesy()`. Returns the number of processors in the X/Y dimension of the mesh.
- `peid()`. Provides the identifier of the processor.
- `peidx/peidy()`. Provides the X/Y axis identifier of the processor in the mesh.
- `pidowner(i,j)`. Returns the identifier of the processor which contains entry *(i,j)*. It is a local operation: *(i mod npesy())*$\times$*npesx() + j mod npesx()*, due to the distribution scheme we are using.
- `pidxowner(j)/pidyowner(i)`. Returns the identifier in the X/Y axis of the processors which contain the entries of column/row *j*/*i*.

There are also functions to convert global indices into local indices (*locrow(i)* and *loccol(j)*) and vice versa (*globrow(i)*/*globcol(j)*).

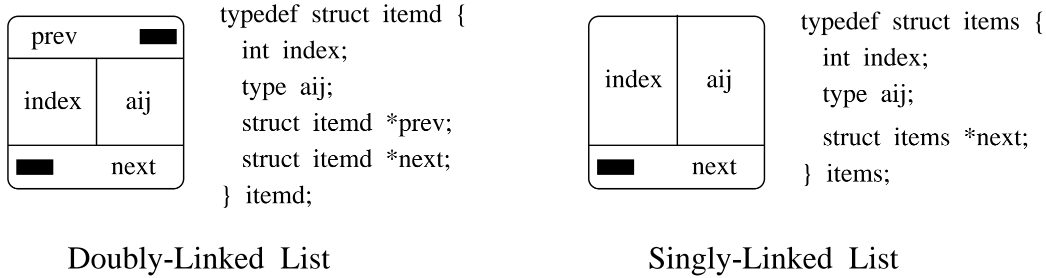

Figure 5.18: *Data structures for list management*

5.6.2 List Management

In order to perform more complex operations, users can combine the routines of the *3LM* library with the manual handling of the linked list data structures. This management requires a deeper knowledge of the underlying data structures of the library. Therefore, the declaration of these structures, shown in Figure 5.18 (for one-dimensional lists) must be taken into account by the user, where *type* represents the int, float or double data types. There are similar data structures for two-dimensional linked lists.

Programmers can access to the pointers to the beginning and to the end of the lists using the functions *lll_dbegin* and *lll_dend*, respectively (*lll_sbegin* and *lll_send* for singly-linked lists):

```
pointerd *lll_dbegin/lll_dend(dll list_id)
pointers *lll_sbegin/lll_send(sll list_id)
```

As we can see, in order to refer to these pointers, we must make use of the predefined data types *pointerd* (for doubly-linked lists) and *pointers* (for singly-linked). Therefore, with these functions we can access the elements of the lists directly. For instance, if we want to display on the screen the nonzero elements of matrix R obtained in the code of Figure 5.11, in a row-column-entry format (coordinate format), we must first define:

```
pointerd *firstr, aux
```

and add the code of Figure 5.19 (note that the condition in the *while* loop is because these lists end with the NULL value, as we show in Figures 5.1 and 5.6).

Similar functions for two-dimensional linked lists are:

```
pointer2d *lll_2dbegin/lll_2dend, pointer2s *lll_2sbegin/lll_2send
```

the first two *3LM* routines for doubly-linked lists and the last ones for singly-linked.

We must take into account that, after using a reorientation procedure for lists (see Subsection 5.2.2), the old values previously returned by the functions

```
firstr=lll_dbegin(listR_id);
for (j=fcol(0); j<fcol(n); j++) {
   aux=firstr[j];
   while (aux) {
      printf ("\n i=%d, j=%d, aij=%le", globrow(aux->index),
              globcol(j), aux->aij);
      aux=aux->next;
     }
  }
```

Figure 5.19: *3LM low-level code*

lll_dbegin, *lll_dend*, etc. are not valid yet, because they have been changed by means of the reorientation function. Therefore, these functions should be applied again to obtain the right values.

5.6.3 User-Defined Functions

As we mentioned in Section 5.3, in addition to the predefined functions *OpAdd, OpSub, OpMul, OpDiv, Nop*, which can be passed as a parameter of the routines *lll_doper, lll_doperv, lll_dfillin*, etc. (and the analogous routines for singly-linked lists and two-dimensional lists), users can define their own set of functions to perform list operations. In order to do this, a binary function must be declared. For instance, if we want to overwrite all the nonzero entries a_{ij} of a matrix A (stored in one-dimensional singly-linked lists) with $7.0 \times \log a_{ij}$ (an operation without fill-in), the user should first declare this function:

```
type OpExample (x, y)
type x, y;
{
 return log10(x)*y;
}
```

and the corresponding *3LM* code (using the routine *lll_soper*) to perform the operation described above is:

```
for (j=fcol(0); j<fcol(n); j++)
   lll_soper(j, listA_id, 7.0, OpExample, frow(0), frow(m));
```

5.7 Comparison Results

It would be interesting to compare the parallel manual algorithms described in Chapter 3 with the ones coded using *3LM* routines. Table 5.1 shows this comparison between the manual MGS algorithm (see Section 3.2) and the corresponding

JPWH991	*Manual code*		*3LM code*		**%Overh.**
	Ex.Time	**Speed-up**	**Ex.Time**	**Speed-up**	
1 PE	241.36s	1.00	805.32s	1.00	234%
2×2 PEs	71.35s	3.38	264.67s	3.04	271%
4×4 PEs	18.98s	12.72	96.97s	8.30	411%
8×8 PEs	7.43s	32.48	40.22s	20.02	441%

ORANI678	*Manual code*		*3LM code*		**%Overh.**
	Ex.Time	**Speed-up**	**Ex.Time**	**Speed-up**	
1 PE	504.42s	1.00	2545.44s	1.00	405%
2×2 PEs	146.09s	3.45	828.23s	3.07	467%
4×4 PEs	50.30s	10.03	291.78s	8.72	480%
8×8 PEs	21.56s	23.40	126.14s	20.18	485%

SHERMAN5	*Manual code*		*3LM code*		**%Overh.**
	Ex.Time	**Speed-up**	**Ex.Time**	**Speed-up**	
1 PE	607.35s	1.00	2897.42s	1.00	377%
2×2 PEs	172.62s	3.52	883.15s	3.28	412%
4×4 PEs	73.62s	8.25	364.46s	7.95	395%
8×8 PEs	26.96s	22.53	148.77s	19.48	452%

Table 5.1: *MGS: execution times and speed-ups for matrices JPWH991, ORANI678 and SHERMAN5*

3LM version presented in Figure 5.11. Execution times and speed-ups are provided for three sparse matrices: *JPWH991*, *ORANI678* and *SHERMAN5* (see Table 2.1). The experiments were conducted on a Cray T3D; therefore, the execution times for the manual MGS algorithm are the same as the ones presented in Table 4.5 (using doubly-linked lists). The field *%Overh.* of Table 5.1 represents the increase in the execution times of the *3LM* algorithm in relation to the manual one. As we can see, there is an important (and obvious) overhead (although these algorithms scale rather well). This is due to the fact that the *3LM* algorithms:

- do not use packed vectors to perform some basic vector operations; their approach is to scatter the corresponding row/column of the sparse matrix (represented by a list) on a dense vector (using *lll_*unpack* routines) to perform these operations (see, for instance, the sparse matrix-sparse matrix sum examples of Subsection 5.5.1).
- do not use fast communication primitives (for example, the ones specific for the Cray T3D supercomputer) in the underlying implementation; this was done for portability reasons.
- do not group communications together; for example, if we want to calculate a reduction operation of each column of a sparse matrix of n columns, we should apply the corresponding reduction routine to each column of the matrix; internally, it involves n reduction operations. But it can be optimized, using low-level features (not used in our algorithms), grouping

the communications of the n reduction operations together, in order to reduce start-up times.

- do not have the deep knowledge of the programmer about the parallel algorithms to optimize them.

In conclusion, the *3LM* library consists of easy to program general purpose routines, suitable for the field of sparse direct factorizations and their applications. Therefore, it is clear that the execution times cannot be competitive compared to the ones obtained using very optimized manual codes, such as the ones described in the previous chapters. But, in contrast, these routines allow developing parallel sparse codes effortlessly; besides, if more efficient codes are required, it offers the chance of combining those routines with the low-level capabilities described in Section 5.6.

Chapter 6

HPF Extensions for Sparse Operations

6.1 Introduction

In recent years, there have been important research efforts in developing efficient parallel numerical codes for distributed memory multiprocessors. The data-parallel paradigm has emerged as one of the most successful programming models. Recently introduced parallel languages, such as CM Fortran [93], Vienna Fortran [22][110], Fortran D [36], Craft [76] and *de facto* standard High-Performance Fortran (HPF) [56][57], follow this approach.

In a data-parallel programming model, the user writes code using a global index space and compiler directives to set data and work distributions. Data-parallel compilers, taking into account this information, reorganize the code, translate global addresses into local addresses and insert communications to execute the program on a multiprocessor following an SPMD model.

All these languages had initially focused on regular computations, that is, well-structured codes that can be efficiently parallelized at compile-time using simple data (and computation) mappings. However, the situation is different for irregular codes, where data-access patterns and workload are usually known only at run-time. In the literature we can recognize two approaches to deal with irregular codes. The first approach, compile-time techniques, extends the parallel language with new constructs (compiler directives) to express non-structured parallelism. With this information, the compiler can perform at compile-time a number of optimizations, usually embedding the rest of them into a run-time library (as we will see in Section 7.2 of the next chapter). In the second approach, run-time techniques, the non-structured parallelism is captured and managed fully at run-time.

Regarding the first approach, Fortran D and Vienna Fortran data-parallel languages, for instance, include some support for irregular data distributions. In Fortran D, programmers can specify a mapping of array elements to processors

using another array. Vienna Fortran, on the other hand, let programmers define functions to specify irregular distributions. However, HPF in its first Forum [56] does not directly support any of these constructs. This fact has been acknowledged by some researchers, proposing a number of extensions to HPF trying to correct this shortcoming [21][73], and by the HPF Forum in its decision for developing the version 2.0 of HPF (HPF-2) [57]. This second release of HPF has been improved providing a generalized block distribution (GEN-BLOCK), where the contiguous array partitions may be of different sizes, and an indirect distribution (INDIRECT), where a mapping array is defined to specify an arbitrary assignment of array elements to processors. Others, however, prefer the second approach, proposing run-time techniques to automatically manage programmer-defined data distributions, partition loop iterations, remap data and generate optimized communication schedules. Most of these solutions are based on the inspector-executor paradigm [18][78][79].

In any case, the current constructs included in these languages and the supportive run-time libraries are insufficiently developed, leading to low efficiencies when they are applied to a broad set of irregular codes, appearing in the majority of real scientific and engineering applications. To contribute to the solution of this problem the Computer Architecture research group at the University of Málaga has developed and extensively tested pseudo-regular data distributions, designed as natural extensions of regular data distributions [6][9][84][100][101]. The aim of these distributions is their simplicity for incorporation to a data-parallel language and be used by a programmer, together with their effectiveness in obtaining high efficiencies from the parallelization of irregular codes. But the underlying data structures do not deal with fill-in computations and, therefore, we will discuss in this chapter the above related issues in the scene of sparse matrix computations involving fill-in and pivoting operations. Sparse direct factorizations (such as QR orthogonalizations), as we have shown in the previous chapters, present this kind of computations.

Henceforth, Fortran will be the implementation language of our algorithms instead of C language, used in the previous chapters.

6.2 A First Data-Parallel Implementation Using Craft

The basis language of Craft [76] is Fortran 77 plus a number of Fortran 90 extensions such as array syntax and intrinsic functions to perform global operations on shared data. It includes a set of compiler directives (beginning with the keyword *CDIR$*) to distribute data and work. Craft is only implemented on Cray Research Massively Parallel Processors. Although we refer to Craft as a data-parallel model, it also supports several other styles of programming, which may be combined within the same program: message-passing, global address (shared data) and work-sharing.

```
CDIR$ DO SHARED(i) on vv(i)
DO i = inf, sup
  v(vi(i)) = vv(i)
END DO

        (a)
```

```
induc = pos
DO i= inf, sup
  IF (v(i) .NE. 0.0) THEN
    vv(induc) = v(i)
    vi(induc) = i
    induc = induc + 1
  END IF
END DO

        (b)
```

Figure 6.1: *Sparse QR Craft example codes*

A hybrid language of Craft and HPF, HPF_Craft [57, Annex D], combines an SPMD execution model with the highest performing of the HPF features. It is being implemented for Cray MPP systems and may also be available on Cray vector architectures. The goal of HPF_Craft is to attain a well-defined extrinsic interface to the standard HPF.

As a first approach to the data-parallel programming style, we tried to implement a simplified sparse MGS algorithm (without column pivoting and without dynamic memory allocation, due to the use of Fortran 77) using Craft on a Cray T3D. Basically, we implemented the corresponding sequential algorithm in Fortran, using a CCS data structure (see Section 2.6) and we dealt with fill-in by means of copying the data structure in each iteration of the algorithm to include the newly generated elements. We detail this approach for the sparse MGS and LU factorizations in [97].

Craft only provides the classical *BLOCK(x)* distribution for dense matrices, where x indicates that each PE receives x contiguous entries of an array, starting on PE #0 (if x=1, we have a cyclic distribution). These distributions are not suitable for sparse matrix computations, as we will show in the QR example code segments of Figure 6.1, which are extracts of our Craft code. The *DO SHARED* directive in Figure 6.1a distributes the work of the loop; it assigns iteration i to the PE which owns *vv(i)*. This piece of code unpacks a compressed vector (with values stored in vector *vv* and row indices in *vi*) into a dense one, v (scatter operation). The assignment statement with indirections on the left-hand side prevents exploiting locality (it is not guaranteed that *v(vi(i)), vi(i)* and *vv(i)* are located in the same processor) because none of the dense data distributions provided by Craft are appropriate for the CCS sparse data structure.

The code in Figure 6.1b performs the opposite procedure (gather operation): it packs the entries of a dense vector with many zero entries, v, into a compressed one (represented by *vv* and *vi*). Craft fails when compiling this piece of code because of the use of an induction variable called *induc* and its increment inside a conditional statement; the execution of these sentences produces erroneous (and undefined) results in the entries of the compressed vector.

```
REAL:: DATA(alpha)
INTEGER:: COL(alpha), ROW(m+1)
!HPF$ PROCESSORS, DIMENSION(P1,P2):: mesh
!HPF$ REAL, DYNAMIC, SPARSE(CRS(DATA,COL,ROW)):: A(m,n)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO mesh:: A
```

Figure 6.2: *HPF specification of the BRS distribution*

6.3 The SPARSE Directive

As can be observed, current data-parallel languages do not provide support to specify data distributions for sparse matrices or flexible data structures for storing these matrices. However, the distribution strategy of a sparse matrix across the processors and the data structure chosen to store the corresponding local sparse matrices is essential to obtain high parallel efficiencies, as shown in the experimental results of Chapter 4 on sparse QR algorithms. An interesting survey on current data-parallel languages, focusing on data distribution features, is presented in [26].

It would be of major interest if we could incorporate to a data-parallel language special data distributions and data structures for sparse computations. Ujaldón et al [101] introduced the *SPARSE* directive to incorporate in Vienna Fortran and HPF the MRD and BRS/BCS pseudo-regular data distributions (see Section 3.1). Although these distributions can be specified using a mapping array, for instance, through the HPF-2 *INDIRECT* directive, these mapping arrays present several drawbacks, such as the amount of memory consumed and the fact that all optimization analysis must be postponed until run-time. Besides, as mapping arrays are usually distributed over the processors, there is also a significant communication overhead whenever a processor needs to locate a data item distributed through this mapping array. However, the pseudo-regular distributions mentioned above are more efficient, as a simple modulus calculation is enough to locate the data items. For instance, an HPF-like specification of the BRS data distribution may be as indicated in Figure 6.2 (see [101] for more details).

The new *SPARSE* directive means, in this figure, that the sparse matrix A is actually represented in a CRS format, using the arrays *DATA*, *COL* and *ROW* (*alpha* is the number of nonzero entries of matrix A). The *DYNAMIC* keyword in the *SPARSE* directive means that the contents of the above three arrays are determined dynamically, as a result of executing a *DISTRIBUTE* statement.

BRS is a cyclic distribution of the compressed representation of a sparse matrix. Stating *CYCLIC* in a *DISTRIBUTE* directive is understood by the compiler as applying a BRS distribution to the *DATA*, *COL* and *ROW* arrays declared in the *SPARSE* statement. This way, we have the benefits of a cyclic data distribution (load balancing and simple data addressing schemes, that lead to simple

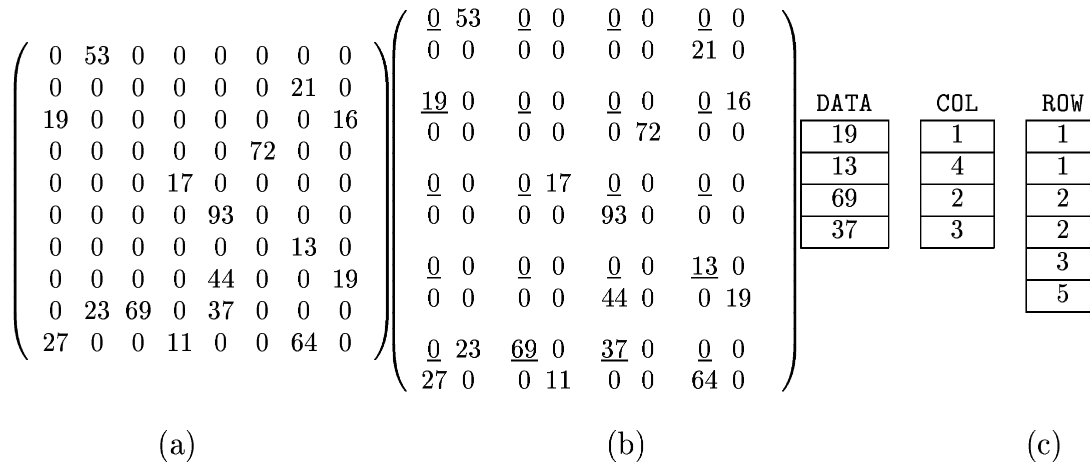

Figure 6.3: *BRS partitioning for a 2×2 processor mesh*

communication patterns) applied to a sparse matrix independently of the compressed format used to represent it. Figure 6.3 shows the result of applying this directive, for $m = 10$, $n = 8$, $P1 = P2 = 2$ and $alpha = 16$. In this example, the sparse matrix of Figure 6.3a is partitioned into a set of submatrices of size 2×2 (Figure 6.3b), which are further mapped, in the same way, onto the processor mesh. As an example, the underlined entries in Figure 6.3b are assigned to processor (0,0) of the mesh. Finally, the sparse local submatrices are stored in a CRS format, as depicted in Figure 6.3c for the processor (0,0).

6.4 HPF-2 Proposals for Sparse Distributions

The tendency in data-parallel languages is to use Fortran, because of its extended utilization in scientific codes and because it is less flexible than C, which makes the implementation of the compiler easier.

We have chosen HPF to propose our data-parallel extensions for two main reasons:

- *Standarization.* HPF is a standard of the High Performance Fortran Forum (HPFF). HPFF is a coalition of industrial and academic groups working to suggest a set of standard extensions to Fortran 90 to provide support for high-performance programming on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors. The last version is HPF 2.0 [57] and it is an extension of the current Fortran standard Fortran 95.

- *Fortran 90 basis.* The basis of the data-parallel extensions described in [101] is Fortran 77 and they do not consider pivoting operations or fill-in. We propose data-parallel extensions for sparse matrix computations with pivoting and fill-in, using Fortran 90 [35] as basis language, due to the fact that

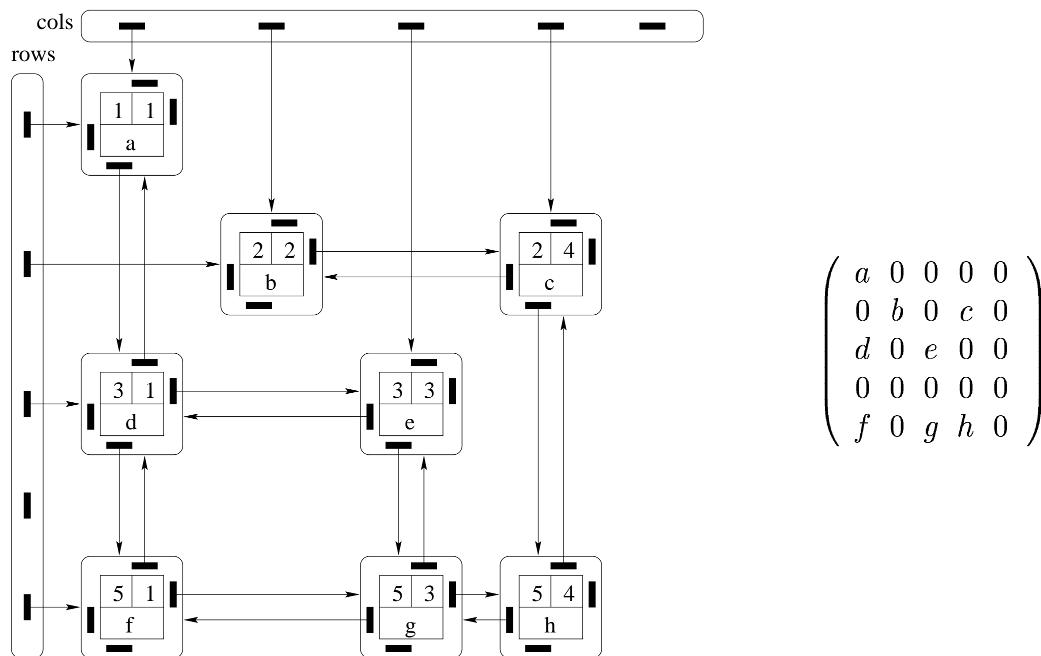

Figure 6.4: *LLRCS data storage scheme*

dynamic data structures (pointers, memory allocation, etc.) are required to support these features.

First, we propose four data storage schemes to support the sparse matrices (or vectors) to be distributed:

- *LLCS*, Linked List Column Storage.
- *LLRS*, Linked List Row Storage.
- *LLRCS*, Linked List Row-Column Storage.
- *CVS*, Compressed Vector Storage.

The first three schemes represent sparse matrices and the last one represents one-dimensional sparse arrays. The *LLCS* storage scheme corresponds to the first structure shown in Figure 2.8, that is, the matrix is represented by columns stored as linked lists (but considering that, from now on, for all our Fortran codes, we assume that the arrays begin in index 1). Observe that in the figure the lists are doubly-linked, but they can also be defined as singly-linked, in order to save memory overhead. The *LLRS* storage scheme is similar to *LLCS*, but considering linking by rows instead of columns. A two-dimensional linked list data structure can be declared using the *LLRCS* storage scheme, as shown in Figure 6.4. As well as with the other two schemes, the entries can be singly or doubly-linked. Finally, the *CVS* scheme represents a sparse array as two arrays and one scalar: the index array, containing the indices of the nonzero entries of the sparse array,

```
!Doubly LLRS, LLCS (one-dimensional)
TYPE entry
  INTEGER:: index
  REAL:: value
  TYPE (entry), POINTER:: prev, next
END TYPE entry

!Doubly LLRCS (two-dimensional)
TYPE entry
  INTEGER:: indexi, indexj
  REAL:: value
  TYPE (entry), POINTER:: previ, prevj, &
                          nexti, nextj
END TYPE entry
```

```
TYPE ptr
  TYPE (entry), POINTER:: p
END TYPE ptr

TYPE (ptr), DIMENSION(n):: pex
```

Figure 6.5: *Fortran 90 derived data types for the list items and declaration of an array of pointers to these items*

the value array, containing the nonzero entries themselves, and a scalar that stores the number of nonzero entries.

The left side of Figure 6.5 displays the Fortran 90 derived data types which define the corresponding items of each kind of linked list. The first type corresponds to the *LLRS* and *LLCS* schemes (doubly-linked), indistinctly, and the second one, to the *LLRCS* scheme (doubly-linked too). The singly-linked versions for these data types are similar, but without the *prev* pointers. The next step consists in defining one array (or two) of pointers to the above items. It is not possible in Fortran 90 to directly declare the array of pointers we wish. But Fortran 90 allows us to declare such array(s) (say *pex*) indirectly through the use of a derived data type (we called it *ptr*), as depicted in the right side of Figure 6.5. Therefore, *pex* is an array of n pointers, each one of them pointing (if we have an LLCS scheme) to one column of the sparse matrix.

Once the storage schemes have been defined, we can use the *SPARSE* directive (see Section 6.3) to specify that a sparse matrix (or sparse array), say A, is stored using a particular linked list scheme. That is, A represents a place holder (template) for the sparse matrix, and the *SPARSE* directive establishes a connection between the logical entity A and its internal representation (linked lists). The benefit of this approach is that we can use the standard HPF *DISTRIBUTE* and *ALIGN* directives applied to the matrix A and, at the same time, store the matrix itself using a compressed format. As an example, and connecting with the preceding chapter, the LLCD (Linked List Column Distribution) scheme of the *3LM* library (shown in Figure 5.1) is equivalent to the LLCS storage scheme of our extended HPF-2 combined with the *DISTRIBUTE(CYCLIC, CYCLIC)* directive.

The *SPARSE* directive described in Section 6.3 can be easily extended to incorporate the new linked list data structures. Figure 6.6 shows a simplified Backus-Naur (*BNF*) syntax [43, Appendix 1] of our proposal for such direc-

```
<sparse-directive>::=<datatype>, SPARSE (<sparse-content>)::<array-objects>
<datatype>::= REAL | INTEGER
<sparse-content>::= LLRS (<ll-spec>)
                  | LLCS (<ll-spec>)
                  | LLRCS (<ll2-spec>)
                  | CVS (<cvs-spec>)
<ll-spec>::= <pointer-array-name>, <pointer-array-name>,
             <size-array-name>,
             <link-spec>
<ll2-spec>::= <pointer-array-name>, <pointer-array-name>,
              <pointer-array-name>, <pointer-array-name>,
              <size-array-name>, <size-array-name>,
              <link-spec>
<cvs-spec>::= <index-array-name>, <value-array-name>, <size-scalar-name>
<link-spec>::= SINGLY | DOUBLY
<array-objects>::= <sized-array>{,<sized-array>}
<sized-array>::= <array-name>(<subscript>[,<subscript>])
```

Figure 6.6: *Syntax for the proposed HPF-2 SPARSE directive*

tive, where the new data structures are the only ones considered. The first two data structures, *LLRS* and *LLCS*, are defined by two arrays of pointers (*<pointer-array-name>*) (with the same type as the *pex* array shown in Figure 6.5), which point to the beginning and to the end, respectively, of each row (or column) list, and a third array (*<size-array-name>*), containing the number of elements per row (for *LLRS*) or per column (for *LLCS*). The option *<link-spec>* specifies the type of linking of the list data structure (singly or doubly). Regarding the *LLRCS* data structure, we have four arrays of pointers which point to the beginning and to the end of each row and each column of the sparse matrix, and two additional arrays storing the number of elements per row and per column, respectively.

As an example of using this directive for the Compressed Vector Storage, the following statement:

```
!HPF$ REAL, DYNAMIC, SPARSE (CVS(vi, vv, sz)):: V(10)
```

declares *V* as a place holder of a one-dimensional sparse array, which occupies no storage. What is really stored are the nonzero entries of the sparse array (*vv*), the corresponding indices (*vi*), and the number of nonzero entries (*sz*). The function of the place holder *V* is to provide an abstract object with which other data objects can be aligned and which can then be distributed.

The meaning of the *DYNAMIC* directive (which is used in all our *SPARSE* directives) is that, in this context, the size of the arrays may change during computations.

```
REAL, DIMENSION(10,10):: A
INTEGER, DIMENSION(10):: vi
REAL, DIMENSION(10):: vv
INTEGER:: sz
!HPF$ PROCESSORS, DIMENSION(2,2):: mesh
!HPF$ REAL, DYNAMIC, SPARSE (CVS(vi, vv, sz)):: V(10)
!HPF$ ALIGN V(:) WITH A(*,:)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO mesh:: A
```

Figure 6.7: *HPF-2 specification of a replicated compressed vector*

The HPF-2 directives *DISTRIBUTE* and *ALIGN* can be applied to our sparse place holders with the same syntax as in the standard. As we noted in Section 6.3, distributing a sparse place holder is equivalent to distributing it as if it were a dense matrix/array. In the case of the *ALIGN* directive, however, the semantics must be slightly different. Let us consider the example code of Figure 6.7. The effect of the *ALIGN* directive within this piece of code is the following: the nonzero entries of V (that is, *vv*) are aligned with the columns of A depending on the positions stored in the array *vi*, and not in the corresponding positions in its own *vv* array (which is the standard semantics for dense arrays). Then, the *DISTRIBUTE* directive replicates the V array over the first dimension of the processor array *mesh*, and distributes it over the second dimension in the same way as the second dimension of matrix A, so that each row of the processor mesh stores a complete copy of V. Observe that in this distribution operation, *vi* is taken as the index array for the entries stored in *vv*. As we can see, in order to replicate an object (V) it must first be aligned with a higher-dimensional object (matrix A); if there is no suitable data object to serve as the align target, a template can be declared for this aim. Figure 6.8 shows the combined effect of alignment/distribution for a particular case (we follow the same PE notation as in the mesh of Figure 5.1).

6.5 Extended HPF-2 Sparse Codes

The use of the *SPARSE* directive establishes a link between the sparse matrix (or array) and its storage scheme. From this point on, we can choose to hide the storage scheme to programmers, and allow them to write the parallel sparse code using dense matrix notations, as we initially proposed in [97].

In this proposal, the compiler would be in charge of translating these dense notations into parallel sparse codes taking into account the storage schemes used. However, this approach implies a great effort in compiler implementation (the feasibility of its design is not clear), as well as the possibility of mixing in the same code place holders (that is, *logical* entities) with *real* arrays. A simple consequence of this fact is that if we compile the program in sequential mode the

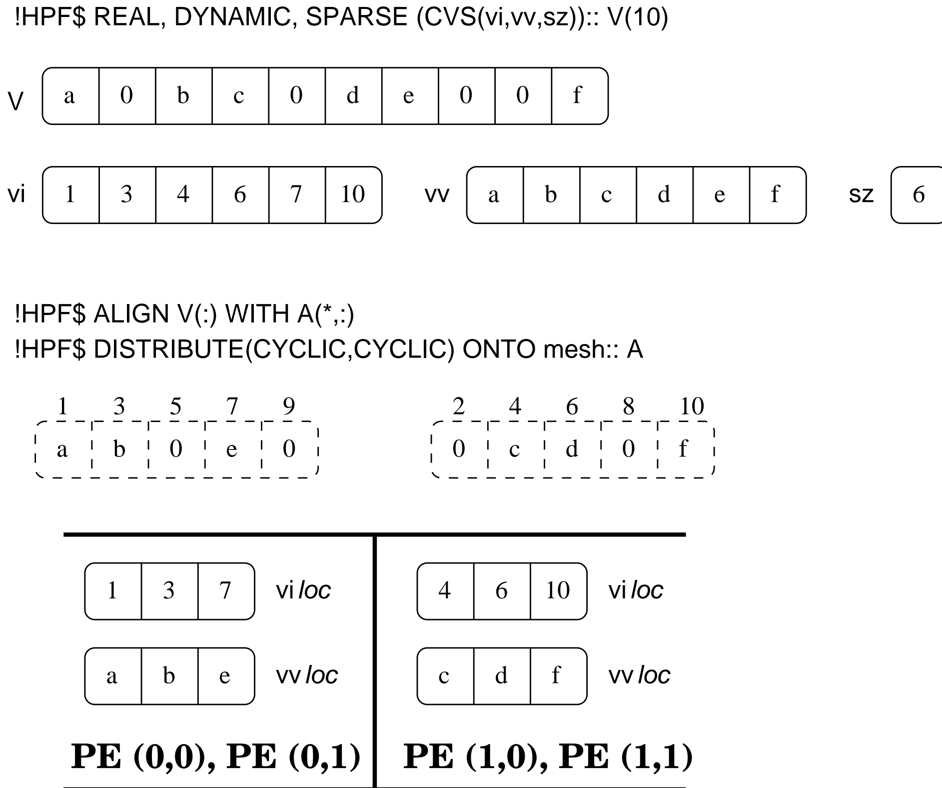

Figure 6.8: *Alignment and distribution of a sparse array on a 2×2 mesh*

result would be erroneous. Bik [13] proposes a similar approach, based on the automatic transformation of a dense program, annotated with sparse directives, into a semantically equivalent sparse code. As a conclusion, this proposal unloads the large part of the work on the compiler and results in easy to program high-level codes. However, the implementation of such a compiler is very complex, and no implementation of it is available for general and real problems.

Therefore, being realistic, as sparse codes are complex, programmers should assume some part of this complexity in their codes, giving hints to the compiler so that it can do a good job. An approach in this line is to force programmers to use the lists and arrays defining the storage scheme explicitly, and allow them to use the place holders only for alignment and distribution purposes. This is the solution employed for the data-parallel sparse QR codes we present next. The same approach was also applied to sparse LU codes in [7].

6.5.1 Sparse MGS Data-Parallel Code

Let us focus on the first of our working example codes, the sparse rank-revealing MGS algorithm (consult Sections 2.2 and 3.2). As we have discussed in Section 2.6, we have chosen one-dimensional doubly-linked lists as the compressed storage format for matrices Q (that is, A, at the beginning of the computation) and R. In this HPF-2 code, we carried out a simple parallelization of the MGS

```
INTEGER, PARAMETER:: m=1000, n=1000, dim=8
REAL, PARAMETER:: accuracy=1.0E-20
!HPF$ PROCESSORS, DIMENSION(dim):: linear
TYPE (ptr), DIMENSION(n):: firstq, lastq, firstr, lastr
TYPE (entry), POINTER:: aux
REAL, DIMENSION(n):: norm, vsum
INTEGER, DIMENSION(n):: vsizeq, vsizer
INTEGER, DIMENSION(m):: vcoli
REAL, DIMENSION(m):: vcolv
INTEGER:: size
INTEGER:: k, i, j, rank, p
REAL:: pivot, product, valmin
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(firstq, lastq, vsizeq, DOUBLY)):: Q(m,n)
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(firstr, lastr, vsizer, DOUBLY)):: R(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(m)
!HPF$ ALIGN norm(:) WITH Q(*,:)
!HPF$ ALIGN vsum(:) WITH Q(*,:)
!HPF$ ALIGN VCOL(:) WITH Q(:,*)
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: Q, R
```

Figure 6.9: *Declaration section of the extended HPF-2 specification of the MGS algorithm*

algorithm by cyclically distributing the columns of the sparse matrices Q and R across a linear array of processors.

Figure 6.9 shows the declaration section of the parallel MGS code using the proposed extensions to HPF-2. Matrices Q and R are defined as sparse and stored using the LLCS data structure. Matrix Q is initially the input matrix A, which is read from a file, while matrix R is computed during the execution of the code. The arrays of pointers *firstq* (*firstr*) and *lastq* (*lastr*) indicate the first and the last nonzero entry, respectively, of each column of the Q (R) sparse matrix. The array *vsizeq* (*vsizer*) stores the number of nonzero entries of each column of Q (R). We have also defined a sparse array *VCOL*, which is stored as a compressed vector (CVS format). This array contains the normalized pivot column of Q (after the pivoting operation), calculated in each outer iteration of the algorithm. The place holders (Q, R and *VCOL*) are in upper-case to distinguish them from the real variables.

The last sentence in the declaration section distributes the columns of both sparse matrices, Q and R, cyclically over a one-dimensional arrangement of processors (declared as *linear* by the *PROCESSORS* directive). Previously, two dense arrays, *norm* and *vsum*, were aligned with the columns of Q. Therefore, after the distribution of Q, these two arrays also appear distributed in a cyclic way over the processors. Finally, the sparse array *VCOL* is aligned with the rows of Q. Hence, after distributing Q, *VCOL* is completely replicated over all the processors. The reason of this replication is that the pivot column is selected and updated in each iteration of the main loop of the algorithm (see loop k of the sequential algorithm in Figure 2.1). This is accomplished by the owner of column k of Q and, hence, the new value must be consistently broadcast. The replication of *VCOL*, which implies this broadcasting operation, exploits data locality.

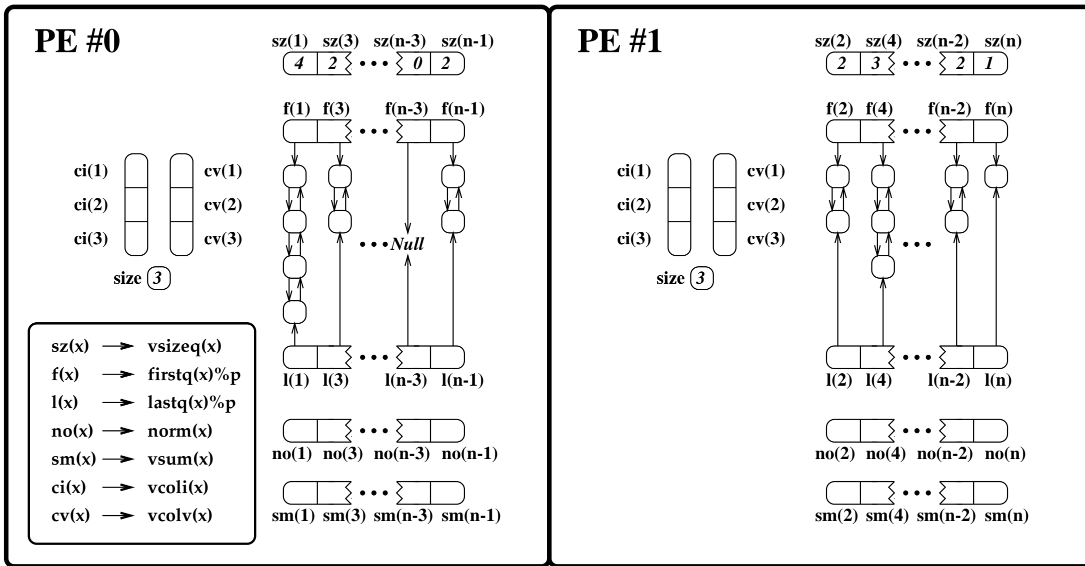

Figure 6.10: *Partitioning of MGS arrays on two processors*

Note that, with this declaration, the linear array of processors is not an essential limitation (a mesh processor could be used). It is just for convenience, as our experiments have produced better results (execution times and efficiencies) for this configuration.

As an example of the above declaration of variables, Figure 6.10 shows the partitioning of the arrays/matrices declared in Figure 6.9 (matrix R is omitted), for the case of two processors (dim = *2*) and an even number of columns for Q. It is also assumed that the outer loop of the MGS algorithm is in the iteration number 4. The legend in the lower left corner of the figure gives the correspondence between the notation employed for naming the data elements and the notation used in the declaration section.

Figure 6.11 presents the rest of the parallel MGS code. First, the norms of the columns of the matrix are calculated and then, a column pivoting, is applied. This column pivoting is based on criterion (2.18) in order to reduce fill-in, and consists in obtaining the pivot column p as the one with the least number of nonzero elements (this number is stored in *valmin*), whenever the norm of this column is greater than the threshold value *accuracy*. As we can see, the pivoting procedure involves a reduction operation and, therefore, we use a *REDUCTION* clause in our HPF code. The effect of this clause is explained in detail in Subsection 6.5.3, because it is also necessary for the least squares problem. A similar approach is used in [7] to program the Markowitz criterion in a sparse LU factorization using our extended HPF-2.

Once the pivot column is selected, the *swap* routine is called to perform the permutation of the current column k and the pivot column p of matrices Q and R.

After computing the squared norms of the columns of A and the pivoting operation, the column k of Q is updated (normalized) and copied into the sparse

```
! --> Norms are calculated
!HPF$ INDEPENDENT, NEW (aux,i)
      DO j = 1, n
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
        aux => firstq(j)%p
        DO i = 1, vsizeq(j)
          norm(j) = norm(j) + aux%value*aux%value;  aux => aux%next
        END DO
!HPF$ END ON
      END DO

      rank = n
! --> Main loop
main: DO k = 1, n
! --> Column pivoting
        valmin = m+1;  p = 0
!HPF$ INDEPENDENT, REDUCTION(valmin, p)
        DO j = k, n
          IF (vsizeq(j) < valmin .AND. norm(j) >= accuracy) THEN
            valmin = vsizeq(j);  p = j
          END IF
        END DO
        IF (p == 0)
          rank = k-1;  EXIT
        END IF
        pivot = norm(p);   norm(p) = norm(k);  norm(k) = pivot
        CALL swap(k, p, firstq, lastq, vsizeq)
        CALL swap(k, p, firstr, lastr, vsizer)
        pivot = SQRT(pivot)

! --> Column k of Q is updated and copied into VCOL
!HPF$ ON HOME (firstq(k)), RESIDENT BEGIN
        aux => firstq(k)%p
        DO i = 1, vsizeq(k)
          aux%value = aux%value / pivot
          vcoli(i) = aux%index;  vcolv(i) = aux%value
          aux => aux%next
        END DO
        size = vsizeq(k)
!HPF$ END ON

! --> Dot-products between column k of Q and columns (k+1:n) of Q
        vsum(k) = pivot;  vsum(k+1:n) = 0.0
!HPF$ INDEPENDENT, NEW (aux,i)
loopj1: DO j = k+1, n
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
          aux => firstq(j)%p
loopi1:   DO i = 1, size
            DO
              IF (.NOT.ASSOCIATED(aux)) EXIT
              IF (aux%index >= vcoli(i)) EXIT
              aux => aux%next
            END DO
            IF (ASSOCIATED(aux)) THEN
              IF (aux%index == vcoli(i)) THEN
                vsum(j) = vsum(j) + aux%value*vcolv(i)
              END IF
            END IF
          END DO loopi1
!HPF$ END ON
        END DO loopj1
                              •
                              •
                              •
```

Figure 6.11: *Outline of an extended HPF-2 specification of the parallel MGS algorithm (first part)*

```
                            ⋮

! --> Subrow (k,k:n) of R is updated
!HPF$ INDEPENDENT
        DO j = k, n
          IF (ABS(vsum(j)) >= accuracy) THEN
            CALL append(k, vsum(j), firstr(j)%p, lastr(j)%p, vsizer(j))
          END IF
        END DO

! --> Submatrix of Q (columns (k+1:n) of Q) is updated
!HPF$ INDEPENDENT, NEW (aux,i,product)
loopj2: DO j = k+1, n
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
          aux => firstq(j)%p
loopi2:   DO i = 1, size
            product = -vsum(j)*vcolv(i)
outer_if:   IF (ABS(product) >= accuracy) THEN
              DO
                IF (.NOT.ASSOCIATED(aux)) EXIT
                IF (aux%index >= vcoli(i)) EXIT
                aux => aux%next
              END DO
inner_if:     IF (ASSOCIATED(aux)) THEN
                IF (aux%index == vcoli(i)) THEN
                  aux%value = aux%value + product
                ELSE
! ----> First or middle position insertion
                  CALL insert(aux, vcoli(i), product, firstq(j)%p, &
                              vsizeq(j))
                END IF
              ELSE inner_if
! ----> End position insertion
                CALL append(vcoli(i), product, firstq(j)%p, &
                            lastq(j)%p, vsizeq(j))
              END IF inner_if
            END IF outer_if
          END DO loopi2
!HPF$ END ON
        END DO loopj2

! --> Norms are updated
        norm(k+1:n) = norm(k+1:n) - vsum(k+1:n)*vsum(k+1:n)

      END DO main
```

Figure 6.11 (cont.): *Outline of an extended HPF-2 specification of the parallel MGS algorithm (last part)*

VCOL array. This is computed by the owner of such column (*ON HOME* directive) and, therefore, as *VCOL* is replicated, the new array is sent to the rest of the processors.

Afterwards, loop j in the MGS algorithm (see Figure 2.1) is computed. This loop is split into three loops: the updating of subrow $R_{k,k:n}$ (the calculation of entry (k,k) is also included), the updating of submatrix $Q_{1:m,k+1:n}$, and the updating of the corresponding squared norms of Q: $norm_{k+1:n}$.

The dot products between column k and columns $(k+1:n)$ of Q are tem-

```
MODULE list_routines
  USE data_structure
  IMPLICIT NONE

CONTAINS

  SUBROUTINE append(ind, product, fptr, lptr, nel)
    INTEGER:: ind, nel
    REAL:: product
    TYPE (entry), POINTER:: fptr, lptr

    TYPE (entry), POINTER:: new
    ALLOCATE(new)
    new%index = ind
    new%value = product
    nel = nel + 1

    IF (.NOT.ASSOCIATED(lptr)) THEN
      NULLIFY(new%prev, new%next)  ! Empty list
      fptr => new
      lptr => new
    ELSE
      new%prev => lptr
      NULLIFY(new%next)
      lptr%next => new
      lptr => new
    END IF
  END SUBROUTINE append

        ...

END MODULE list_routines
```

Figure 6.12: *Fortran 90 module for list management*

porarily stored in the dense array *vsum*. In this part of the parallel code two *ON HOME (...), RESIDENT* directives were also included, to facilitate the compiler's analysis and code generation tasks. For instance, the *!HPF$ ON HOME (firstq(j))* directive means that the processor which owns the program variable *firstq(j)* should execute the computation. The aim of the *RESIDENT* clause is to indicate that all references to all variables referenced during the execution of the directive's body are local, that is, focusing on our example, all these variables are stored in the local memory of the active processor that owns variable *firstq(j)*. This way, the compiler can use this information to avoid generating communications or to simplify array address calculations, which optimizes the generated code.

The parallel loops were also annotated with the *INDEPENDENT, NEW (...)* directive, with the same meaning as in the standard HPF-2. The *INDEPENDENT* directive indicates that the iterations of the DO-loop which it precedes are independent and they can be executed in any order (and therefore, concurrently), without changing the semantics of the loop. The *NEW* clause of the *INDEPENDENT* directive specifies that the variables indicated in parentheses must be considered private to each iteration, in order to make the iterations

```
INTEGER, PARAMETER:: m=1000, n=1000, dim=8
REAL, PARAMETER:: accuracy=1.0E-20
!HPF$ PROCESSORS, DIMENSION(dim):: linear
TYPE (ptr), DIMENSION(n):: first, last, rowk
TYPE (entry), POINTER:: aux, aux2
REAL, DIMENSION(n):: norm, w
INTEGER, DIMENSION(n):: vsize
INTEGER, DIMENSION(m):: vi
REAL, DIMENSION(m):: vv
INTEGER:: size
INTEGER:: k, i, j, rank, p
REAL:: pivot, product, temp, beta, dotp, valmin
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first, last, vsize, DOUBLY)):: A(m,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vi, vv, size)):: V(m)
!HPF$ ALIGN norm(:) WITH A(*,:)
!HPF$ ALIGN w(:) WITH A(*,:)
!HPF$ ALIGN rowk(:) WITH A(*,:)
!HPF$ ALIGN V(:) WITH A(:,*)
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
```

Figure 6.13: *Declaration section of the extended HPF-2 specification of the Householder algorithm*

independent. Therefore, each iteration must be given a new and independent copy of these variables, which are undefined at the beginning of the iteration and become undefined again at the end.

From the structure of this code we note that if it is converted by a Fortran 90 compiler (that is, the HPF-2 directives are taken as comments), the code runs properly in sequential mode. The code also contains the user routines *append()* and *insert()* for list management, which are included in a Fortran 90 module. For instance, Figure 6.12 shows the code for the routine *append()*, which adds an entry at the end of a list. The *insert()* routine adds an element at the beginning or in the middle of a list. This module requires another module called *data_structure* (not shown in the figure), which includes the declaration of the derived data types (previously described in Section 6.4) to support the linked list data structures.

6.5.2 Sparse Householder Data-Parallel Code

The Householder algorithm (consult Sections 2.3 and 3.3) can also be expressed by means of our extended HPF compiler. The declaration section of this algorithm is presented in Figure 6.13. The interpretation of this code is basically the same as for MGS: the algorithm is mapped onto an array of *dim* processors. In this case, we have only one matrix in LLCS format (A), which is overwritten in-place by matrix R. The Householder vector V is stored in CVS format to reduce the number of computations when performing matrix-vector operations and, by means of the *ALIGN* directive, it is replicated on each processor to favour local computations (it is similar to the *VCOL* vector of the MGS algorithm).

The purpose of the pointer array *rowk* is, basically, to point to the row k of

```
! --> Norms are calculated
      ...

! --> Main loop
      rank = n
main: DO k = 1, n

! --> Column pivoting
      ...

! --> Householder vector is copied in V (stored in CVS format)
! --> Only the owner processor executes this piece of code
!HPF$ ON HOME (rowk(k)), RESIDENT BEGIN
        aux => rowk(k)%p
        temp = 0.0
        IF (ASSOCIATED(aux)) THEN
          IF (k == aux%index) THEN
            IF (aux%value < 0.0) THEN
              pivot = -pivot
            END IF
            temp = aux%value
            aux => aux%next
          END IF
        END IF
        beta = temp+pivot
        vi(1) = k
        vv(1) = 1.0
        size = vsize(k)
        IF (rowk(k)%p%index /= k) size = size+1
        DO i = 2, size
          vi(i) = aux%index
          vv(i) = aux%value/beta
          aux => aux%next
        END DO
!HPF$ END ON

! --> Dot-products between Householder vector and columns (k:n) of A
        w(k:n) = 0.0
!HPF$ INDEPENDENT, NEW (aux,i)
loopj1: DO j = k, n
!HPF$ ON HOME (rowk(j)), RESIDENT BEGIN
          aux => rowk(j)%p
loopi1:   DO i = 1, size
            DO
              IF (.NOT.ASSOCIATED(aux)) EXIT
              IF (aux%index >= vi(i)) EXIT
              aux => aux%next
            END DO
            IF (ASSOCIATED(aux)) THEN
              IF (aux%index == vi(i)) THEN
                w(j) = w(j) + aux%value*vv(i)
              END IF
            END IF
          END DO loopi1
!HPF$ END ON
        END DO loopj1
                                  •
                                  •
                                  •
```

Figure 6.14: *Outline of an extended HPF-2 specification of the parallel Householder algorithm (first part)*

```
                                •
                                •
                                •

        dotp = -2.0/DOT_PRODUCT(vv(1:size), vv(1:size))
        w(k:n) = dotp * w(k:n)
! --> Householder reflection
!HPF$ INDEPENDENT, NEW (aux,aux2,i,product)
loopj2: DO j = k, n
!HPF$ ON HOME (rowk(j)), RESIDENT BEGIN
          aux => rowk(j)%p
loopi2:   DO i = 1, size
            product = w(j) * vv(i)
outer_if:   IF (ABS(product) >= accuracy) THEN
              DO
                IF (.NOT.ASSOCIATED(aux)) EXIT
                IF (aux%index >= vi(i)) EXIT
                aux => aux%next
              END DO
inner_if1:    IF (ASSOCIATED(aux)) THEN
inner_if2:      IF (aux%index == vi(i)) THEN
                  aux%value = aux%value + product
                  IF (ABS(aux%value) < accuracy) THEN
                    aux2 => aux%next
                    IF (aux%index == rowk(j)%p%index) rowk(j)%p => aux2
! ----> Deletion of an entry of the list
                    CALL delete(aux, first(j)%p, last(j)%p, vsize(j))
                    aux => aux2
                  ENDIF
                ELSE inner_if2
! ----> First or middle position insertion
                  CALL insert(aux, vi(i), product, first(j)%p, vsize(j))
                  IF (rowk(j)%p%index > aux%prev%index) &
                    rowk(j)%p => aux%prev
                END IF inner_if2
              ELSE inner_if1
! ----> End position insertion
                CALL append(vi(i), product, first(j)%p, last(j)%p, &
                            vsize(j))
                IF (.NOT.ASSOCIATED(rowk(j)%p)) rowk(j)%p=>last(j)%p
              END IF inner_if1
            END IF outer_if
          END DO loopi2
!HPF$ END ON
        END DO loopj2

! --> Norms, rowk and vsize are updated
!HPF$ INDEPENDENT, NEW (aux)
        DO j = k+1, n
!HPF$ ON HOME (rowk(j)), RESIDENT BEGIN
          aux => rowk(j)%p
          IF (ASSOCIATED(aux)) THEN
            IF (aux%index == k) THEN
              norm(j) = norm(j) - aux%value*aux%value
              rowk(j)%p => aux%next
              vsize(j) = vsize(j)-1
            END IF
          END IF
!HPF$ END ON
        END DO

      END DO main
```

Figure 6.14 (cont.): *Outline of an extended HPF-2 specification of the parallel Householder algorithm (last part)*

```
REAL:: acum
TYPE (ptr), DIMENSION(n):: vaux
REAL, DIMENSION(n):: x, qtb
REAL, DIMENSION(m):: b
!HPF$ ALIGN vaux(:) WITH Q(*,:)
!HPF$ ALIGN x(:)    WITH Q(*,:)
!HPF$ ALIGN qtb(:)  WITH Q(*,:)
!HPF$ ALIGN b(:)    WITH Q(:,*)
```

Figure 6.15: *Extended declaration section for the LSP through MGS*

the matrix stored in linked lists. This array is similar to the auxiliary array *auxp* used in Givens rotations (see Figure 3.6) but, unlike *auxp* (which traverses the lists from bottom to top), *rowk* traverses the lists from top to bottom. Specifically, each entry j of *rowk* points to element A_{kj}, if it is nonzero. If this element is zero, it points to element A_{zj}, being z the closest integer to k, $z > k$, such that $A_{zj} \neq 0$; if no integer z satisfies these conditions, entry j of *rowk* is nullified. Besides, *rowk* is not used to have row access to the list structure; this array is necessary, in each iteration k (see the iteration space of the Householder algorithm in Figure 2.5) that updates submatrix S (described in Section 2.3), to know the beginning of this submatrix represented by lists.

All the comments about the HPF-2 MGS code of the preceding subsection are applicable to the corresponding HPF-2 Householder code of Figure 6.14. In this executable section, the norm calculation, as well as the pivoting procedure are omitted (they were commented for MGS). In the code we have another routine called *delete*, which erases the entry of the list pointed (following the code) by *aux*.

The reason of the updating of the array *vsize* at the end of the code in the last part of Figure 6.14 is because, for the Householder algorithm, entry $vsize_j$ is used to store the number of nonzero elements of subcolumn $A_{k:m,j}$ (following Fortran index notation), in order to apply the pivoting strategy described in Subsection 2.7.1.

6.5.3 Least Squares Problems

As a practical example, we have also implemented the least squares problem through the MGS QR factorization (it would be very similar for the Householder algorithm), using HPF-2 code. The parallel algorithm is, basically, the one described in Section 3.5, but note that the HPF-2 code is for a linear array of processors.

In order to obtain an adequate data distribution for the arrays involved in the least squares problem, we must extend the declaration section of Figure 6.9 with the statements of Figure 6.15. In this figure, four additional vectors are declared (we employed the same notation as in Section 3.5): the right-hand side

```
! --> Init x and vaux
      x = 0.0
!HPF$ INDEPENDENT
      DO j = 1, n
        vaux(j)%p => lastr(j)%p
      END DO

! --> Back-substitution
      DO i = rank, 1, -1
        acum = 0.0
!HPF$ INDEPENDENT, REDUCTION(acum)
        DO j = i+1, rank
          IF (ASSOCIATED(vaux(j)%p)) THEN
            IF (vaux(j)%p%index == i) THEN
              acum = acum + vaux(j)%p%value * x(j)
              vaux(j)%p => vaux(j)%p%prev
            END IF
          END IF
        END DO
        x(i) = (qtb(i)-acum)/vaux(i)%p%value
        vaux(i)%p => vaux(i)%p%prev
      END DO
```

Figure 6.16: *HPF-2 back-substitution for the LSP*

vector b, the vector which stores the product $Q^T b$ (qtb), the solution vector x, and the array of pointers *vaux*. This last one is the auxiliary array used to provide row access during the back-substitution to the column-oriented data structure in which matrix R is stored.

Vector *qtb* (which is initially zeroed) is easily obtained by adding the following statement as the sentence before last of loop i which updates column k of matrix Q (see Figure 6.11):

```
qtb(k) = qtb(k) + aux%value * b(aux%index)
```

Once the MGS factorization ends, we solve the corresponding upper triangular system by means of a back-substitution (see Section 3.5). Figure 6.16 shows the HPF-2 sparse code for the back-substitution described (in dense notation) in expression (3.2). As we can see in this figure, the external loop i is inherently sequential and, therefore, we cannot use an *INDEPENDENT* directive for it. The inner loop j, however, consists in performing a reduction operation on the variable *acum*, and every bit of the reduction can be executed independently and in parallel in each processor. Thus, we annotate the inner loop with an *INDEPENDENT* directive and a *REDUCTION* clause (which also appears in the pivoting procedure) to inform the compiler about this situation. This clause asserts that variable *acum* is updated in the *INDEPENDENT* loop by a series of commutative/associative operations (sum in this case). The value of the reduction variable *acum* after the loop is the result of a reduction tree. The reduction operation of loop j is indeed a user-defined operation and is not considered by the HPF-2 standard, but its inclusion would not add any significant complexity to the

compiler implementation. There also exists some research on detecting complex parallel reductions [91], where IF constructs can appear inside a reduction loop.

Regarding the permutation code section, it is not a parallelizable stage, as we said in Section 3.5. The HPF-2 code for this phase would simply be the corresponding Fortran 90 sequential code.

Chapter 7

Compiler and Run-Time Support

7.1 Compiler Support

The HPF-2 extensions for sparse computations proposed in the previous chapter must be supported by a compiler module, which should be correctly embedded and integrated in an HPF-2 compiler, avoiding collateral effects. This module should carry out, as a broad outline, the following stages: source code analysis, code parallelization and SPMD code generation.

We have chosen the *Cocktail* toolbox [46] for the design of this compiler module, due to the high quality and reliability of the compilers generated with this application. It is composed of a set of tools, largely independent of each other, to design the different stages in the compiler construction: a scanner and a parser read the source, check the concrete syntax, and construct an abstract syntax tree (stored internally as linked records); semantic analysis is performed on the abstract syntax tree; afterwards the abstract syntax tree is transformed into an intermediate representation and, finally, the code generator produces the machine code. These tools accept as input a specification written in a specific language to the tool and generate files in a target language (C or Modula-2).

7.1.1 Source Code Analysis

The lexical analysis of the extended HPF-2 code is the first stage of the compiler module. This analysis deals with those elements of the code specific for sparse matrices. Therefore, the characters (letters, digits and symbols) which make up the new sparse directives, are grouped together constituting lexical components or tokens; for instance, the set of characters 'LLRCS' is a token that is identified as a keyword of the language.

Rex (*Regular Expression Tool*) is the scanner generator of *Cocktail.* It combines the powerful specification method of regular expressions with the generation of highly efficient scanners (five times faster than *Lex* [2] generated scanners). The *Rex* specification consists of a set of regular expressions associated with semantic

```
SCANNER shpf

GLOBAL {
 void shpf_ErrorAttribute
   (yyToken, yyRepairAttribute) short yyToken;
   shpf_tScanAttribute * yyRepairAttribute;
   {}
}

DEFINE Letter = {A-Z a-z} .
       Digit  = {0-9} .
       Identifier  = Letter (Letter | Digit | "_") * .
       Int_constant = Digit+ .
       A            = {Aa} .
       B            = {Bb} .
       C            = {Cc} .

               •
               •
               •

       X            = {Xx} .
       Y            = {Yy} .
       Z            = {Zz} .


RULE

"(": {return 2; }
")": {return 3; }
",": {return 4; }
"::":{return 5; }
"+": {return 50;}
"-": {return 51;}
"*": {return 52;}
"/": {return 53;}
"**":{return 54;}
!H P F $:          {return 15;}
D Y N A M I C :    {return 16;}
D I M E N S I O N :{return 17;}
R E A L :          {return 40;}
I N T E G E R :    {return 41;}
S P A R S E :      {return 10;}
L L R S :          {return 20;}
L L C S :          {return 21;}
L L R C S :        {return 22;}
C V S :            {return 23;}
S I N G L Y :      {return 30;}
D O U B L Y :      {return 31;}
Identifier:        {return 1; }
Int_constant:      {return 6; }
```

Figure 7.1: *Scanner specification for the SPARSE directive*

actions (written in one of the target languages: C, Modula-2), which are executed when regular expressions are matching; consult [45] for a complete description of the syntax. The generated scanners are implemented as table-driven deterministic finite automatons. *Rex* automatically computes the line and column position of every token and it can also handle ambiguous specifications.

As an example, Figure 7.1 shows the *Rex* scanner specification for the HPF-2

```
SCANNER shpf PARSER phpf

TOKEN

  Identifier    = 1
  '('           = 2
  ')'           = 3
  ','           = 4
  '::'          = 5
  Int_constant  = 6
  '!HPF$'       = 15
  'DYNAMIC'     = 16
  'DIMENSION'   = 17
  'SPARSE'      = 10
  'LLRS'        = 20
  'LLCS'        = 21
  'LLRCS'       = 22
  'CVS'         = 23
  'SINGLY'      = 30
  'DOUBLY'      = 31
  'REAL'        = 40
  'INTEGER'     = 41
  '+'           = 50
  '-'           = 51
  '*'           = 52
  '/'           = 53
  '**'          = 54


OPER

  LEFT '+'  '-'
  LEFT '*' '/'
  RIGHT '**'
  LEFT unary_defined_operator

               •
               •
               •
```

Figure 7.2: *Parser specification for the SPARSE directive (first part)*

SPARSE directive proposed in Section 6.4. Only the indispensable symbols and keywords were included. Note that, following Fortran 90 syntax, upper-case and lower-case letters are treated as identical in both keywords and identifiers.

The syntactic analysis performed by a parser determines if the stream of tokens provided by the scanner can be generated by a certain grammar. In order to carry out this analysis, a syntactic tree associated with the tokens of the source code is constructed. This data structure contains all the syntactic information of the source code and allows fast findings and extractions of information about the code; besides, subsequent tranformations on the code can be easily applied.

Cocktail offers the chance of selecting among three different parser generators (*Lalr, Ell* and *Lark* [47][44]), which transform a grammar into a parser. We have chosen the first one, which processes the class of *LALR(1)* (lookahead-LR) gram-

```
                          .
                          .
                          .

RULE

hpf_directive: hpf_header sparse_directive '::' array_objects
             .
hpf_header: '!HPF$' datatype
          .
datatype: 'REAL'
        | 'INTEGER'
        .
sparse_directive: sparse_stmt
                | sparse_stmt opt_dyn
                | sparse_stmt opt_dim
                | sparse_stmt opt_dyn opt_dim
                | sparse_stmt opt_dim opt_dyn
                | opt_dyn sparse_stmt
                | opt_dim sparse_stmt
                | opt_dyn opt_dim sparse_stmt
                | opt_dim opt_dyn sparse_stmt
                | opt_dyn sparse_stmt opt_dim
                | opt_dim sparse_stmt opt_dyn
                .
array_objects: (Identifier [array_content]) || ','
             .
array_content: '(' int_expr [',' int_expr] ')'
             .
opt_dyn: ',' 'DYNAMIC'
       .
opt_dim: ',' 'DIMENSION' array_content
       .
int_expr: '(' int_expr ')'
        | int_expr '+' int_expr
        | int_expr '-' int_expr
        | int_expr '*' int_expr
        | int_expr '/' int_expr
        | int_expr '**' int_expr
        | '-' int_expr  PREC unary_defined_operator
        | Int_constant
        | Identifier
        .
sparse_stmt: ',' 'SPARSE' '(' sparse_content ')'
           .
sparse_content: 'LLRS'  '(' ll_spec  ')'
              | 'LLCS'  '(' ll_spec  ')'
              | 'LLRCS' '(' ll2_spec ')'
              | 'CVS'   '(' cvs_spec ')'
              .
cvs_spec: Identifier ',' Identifier ',' Identifier
        .
ll_spec: cvs_spec ',' link_spec
       .
ll2_spec: cvs_spec ',' cvs_spec ',' link_spec
        .
link_spec: 'SINGLY'
         | 'DOUBLY'
         .
```

Figure 7.2 (cont.): *Parser specification for the SPARSE directive (last part)*

mars [2]. Syntactic errors are handled completely automatically by the generated parser, including error reporting, error recovery and error repair, following the

complete backtrack-free method. The parser created by *Lalr*, which consists in a stack automaton controlled by a parse table, is two to three times faster than the one produced by *Yacc* [2]. As in the regular expressions of the scanner, each grammar rule may be associated with a semantic action, which is executed when the corresponding rule is recognized by the table-driven parser generated.

Lalr directly accepts only grammar rules in plain *BNF* notation [43, Appendix 1]. But we wrote the grammar rules using *EBNF* (Extended *BNF*) constructs and converted it to plain *BNF* using a grammar transformer provided by *Cocktail* (see [47] for further details).

The *Lalr* parser specification of the *SPARSE* directive is shown in Figure 7.2. From an implementation point of view, it should extend the parser specification of an HPF standard compiler in order to include the new directive. As we can see, the scanner generated by *Rex* (called *shpf*) offers an interface to be used by the parser that will be created by *Lalr*. The *TOKEN* section defines the terminals of the grammar and their encoding. In the *OPER* section, precedence and associativity of operators are specified to resolve ambiguities (LR-conflicts); it is necessary for the integer expressions that can appear in a *DIMENSION* attribute of the *SPARSE* directive (as we can see, the *DIMENSION* attribute, in this context, does not allow subarray notation [35]). The *RULE* section contains the grammar rules and semantic actions (which were omitted to simplify). A complete definition of the specification language can be found in the corresponding user manual [47].

7.1.2 Parallelization and Code Generation

Once the source code analysis is completed, the parallelization stage begins. It consists in extracting the information of the syntax tree generated in the analysis stage in order to carry out the adequate parallelization strategies. The main part of this information is obtained in the compiler directives provided by the user.

The parallelization phase focuses on the following goals:

- Generation of the sparse data structures and their distribution/alignment on the processors.
- Loop parallelization: distribution of the load (loop iterations) related to sparse computations among the processors, generally following the owner-computes rule (each iteration is executed on the owner processor of the referenced data in the left-hand side of an assignment statement) or following the guidelines of a possible *ON HOME* directive.
- Translation of the global indices of the sparse structures into local indices.
- Management of reduction operations involving sparse data structures. These operations entail a strategy of computation distribution so as to exploit the intrinsic locality expressed by the corresponding *REDUCE* clause.

A more complex analysis stage would allow grouping communications together to reduce start-up times, overlap communications and computations and detect user-defined reduction operations, even with IF constructs (the *REDUCTION* clause was discussed in Subsection 6.5.3).

The final step, the code generation stage, generates SPMD (Single Program Multiple Data) Fortran 90 code, extended with calls to run-time standard message-passing routines to access non-local data, so that the portability of the final code is guaranteed. Code transformations are carried out to insert the run-time support so as to perform the tasks associated with the previous parallelization stage.

This SPMD code will be accepted by a standard Fortran 90 compiler of the target parallel machine to be translated into an executable binary code for that machine.

7.2 Run-Time Support

As we saw in the previous section, the HPF compiler acts as a source-to-source preprocessor, generating SPMD Fortran 90 source code. This code includes run-time routines to perform the corresponding parallelization and communication tasks. In order to support our HPF-2 extensions for sparse matrices, we have extended the run-time library *DDLY* (Data Distribution Layer) [99] [100] with a set of routines to handle linked lists and compressed vectors. These routines are called from the SPMD code generated by our extended HPF-2 compiler.

Basically, *DDLY* performs a set of operations for the distribution and alignment of arrays (dense and sparse) on distributed memory multiprocessors. Specifically, it provides an interface which allows managing arrays as abstract objects, hiding from the user the internal data structures used to store and distribute data. It also covers all necessary message-passing handling and optimization, and input/output management. The user interface to *DDLY* is based on object descriptors, which store information about the corresponding object. A basic operation consists in defining an array/matrix descriptor and assigning it to a particular data object. These descriptors are managed in a similar way to the management of file descriptors in UNIX. The assignment of a descriptor to a data object is accomplished by calling a special *open* function. From this step on, every manipulation of the data object is performed through the corresponding descriptor. For instance, the library includes a number of procedures to distribute and align data objects. We only need to specify the descriptor of the object (array/matrix) and some other arguments to perform the desired operation. With this interface, each HPF-2 distribution/alignment directive is translated into a corresponding call to one or several *DDLY* functions. As desired, *DDLY* follows an SPMD programming paradigm, that is, the library functions are called by all the processors with the same arguments. Therefore, the functions are executed in a coordinated way.

In Chapter 5, we described the *3LM* library. As we said, it is based on the *DDLY* library because they both are libraries for sparse computations, they use descriptors (called identifiers in the *3LM* library) for representing data objects and follow an SPMD model. But the *3LM* library has higher-level routines than *DDLY* (although *3LM* includes low-level features, as described in Section 5.6). That is, the *DDLY* routines are suitable for implementing a parallel compiler, whereas the *3LM* library is more adequate to be used by a common programmer of specific parallel sparse algorithms.

As an example of the code generated with *DDLY* routines, Figures 7.3 and 7.4 show a hypothetical SPMD output of the extended HPF-2 compiler described in Section 7.1. The selected example is the sparse MGS algorithm, whose corresponding HPF-2 code was displayed in Figures 6.9 and 6.11. The calculation of the associated least squares problem was also included (see HPF code in Subsection 6.5.3).

7.2.1 Declaration Section

Figure 7.3 shows the hypothetical output of the extended HPF-2 compiler for the declaration section of the MGS algorithm presented in Figures 6.9 and 6.15.

First, the procedure *ddly_new_topology* initializes the desired topology, in which the SPMD code will be executed: a linear array of *dim* processors, whose characteristics are stored in the topology descriptor *td*. The effect of the procedure *ddly_HB_read* is that a sparse matrix and a vector are read from a file in Harwell-Boeing format [34]. The sparse matrix is stored in a compressed format (see Section 2.6) using the triplet of vectors: *data, row, col* and the vector is stored in a dense array *b*.

By means of the *ddly_new* routine, matrix Q is associated to a matrix descriptor *md_Q*, which indicates that sparse matrix Q is stored in a compressed format by columns (*CCS*) and the data type of its entries is *real* (*DDLY_MF_REAL*); *ddly_init_matrix* initializes matrix Q with the contents of the three vectors (*data, row, col*) previously read from a file. Similar procedures are used for matrix R, except *ddly_HB_read*, as matrix R is generated at run-time (it is initially an empty data structure).

Regarding the compressed vector *VCOL*, the process is the same as for sparse matrices: *ddly_new* associates vector *VCOL* (stored in CVS format and with floating point entries) to a vector descriptor *vd_VCOL* and is initialized by means of the *ddly_init_cvs* routine with its components *vcoli*, *vcolv* and *size* (which are initially empty). Similar calls are carried out for all the arrays declared (*norm, vsum, b*, etc.) in order to associate all of them to a vector descriptor to perform further distribution/alignment operations. Note that vector *b* was initialized with the contents read in the *ddly_HB_read* procedure.

Once the data structures have been created and initialized, the next stage is the distribution/alignment of these structures, using their descriptors. The procedure *ddly_bcs* distributes matrix Q (identified by the descriptor *md_Q*) on

```
!HPF$ PROCESSORS, DIMENSION(dim):: linear
! --> td is a topology descriptor for a (1 x dim) mesh
      CALL ddly_new_topology(td, 1, dim)

!HPF$ REAL, DYNAMIC, SPARSE(LLCS(firstq, lastq, vsizeq, DOUBLY)):: Q(m,n)
! --> A Harwell-Boeing matrix (data, row, col) is read
      CALL ddly_HB_read(n, alpha, data, row, col, b)
! --> A matrix descriptor for Q, md_Q, is created using CCS
      CALL ddly_new(md_Q, CCS, DDLY_MF_REAL)
! --> md_Q is initialized
      CALL ddly_init_matrix(md_Q, n, alpha, data, row, col)

!HPF$ REAL, DYNAMIC, SPARSE(LLCS(firstr, lastr, vsizer, DOUBLY)):: R(n,n)
! --> A matrix descriptor md_R is created and initialized
      ...

!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(m)
! --> An array descriptor for VCOL, vd_VCOL, is created using CVS
      CALL ddly_new(vd_VCOL, CVS, DDLY_VF_REAL)
! --> vd_VCOL is initialized
      CALL ddly_init_cvs(vd_VCOL, vcoli, vcolv, size)


! --> Vector descriptors for all declared arrays are created and initialized
      ...


!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: Q, R
! --> BCS distribution of Q (specified by md_Q)
      CALL ddly_bcs(md_Q, td)
! --> md_Q is now the descriptor of the distributed matrix
! --> CCS data storage is changed into LLCS
      CALL ddly_ccs_to_llcs(md_Q, firstq, lastq, vsizeq, DOUBLY)


! --> Similar calls for matrix R
      ...


!HPF$ ALIGN norm(:) WITH Q(*,:)
! --> norm aligned with the 2nd dimension of Q
      CALL ddly_alignv(vd_norm, md_Q, SecondDim)

!HPF$ ALIGN vsum(:) WITH Q(*,:)
      CALL ddly_alignv(vd_vsum, md_Q, SecondDim)

!HPF$ ALIGN VCOL(:) WITH Q(:,*)
! --> VCOL aligned with the 1st dimension of Q
      CALL ddly_aligncvs(vd_VCOL, md_Q, FirstDim)

!HPF$ ALIGN vaux(:) WITH Q(*,:)
!HPF$ ALIGN x(:)    WITH Q(*,:)
!HPF$ ALIGN qtb(:)  WITH Q(*,:)
! --> vaux, x, qtb aligned with the 2nd dimension of Q
      ...

!HPF$ ALIGN b(:)    WITH Q(:,*)
! --> b aligned with the 1st dimension of Q
      CALL ddly_alignv(vd_b, md_Q, FirstDim)
```

Figure 7.3: *Output of the HPF-2 compiler for the declaration section of the MGS algorithm*

```
! --> Norms are calculated
!HPF$ INDEPENDENT, NEW (aux,i)
! --> Loop is partitioned
      DO j = ddly_LowBound(1), ddly_UpBound(n)
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
          ! ****** ON HOME body ******
!HPF$ END ON
      END DO

      rank = n
! --> Main loop
main: DO k = 1, n
! --> Column pivoting
        valmin = m+1;  p = 0
!HPF$ INDEPENDENT, REDUCTION(valmin, p)
! --> Loop is partitioned: local reduction
! --> (After this reduction, p stores the local index of the pivot column)
        DO j = ddly_LowBound(k), ddly_UpBound(n)
          IF (vsizeq(j) < valmin .AND. norm(j) >= accuracy) THEN
            valmin = vsizeq(j);  p = j
          END IF
        END DO
! --> Global reduction
! --> (After this reduction, p stores the global index of the pivot column)
        p = ddly_ReduceLocMinAbs(valmin, p)
        IF (p == 0)
          rank = k-1;  EXIT
        END IF
        pivot = norm(ddly_LowBound(p):ddly_UpBound(p));
        norm(ddly_LowBound(p):ddly_UpBound(p)) =  &
                 norm(ddly_LowBound(k):ddly_UpBound(k))
        norm(ddly_LowBound(k):ddly_UpBound(k)) = pivot
        CALL swap(k, p, firstq, lastq, vsizeq)
        CALL swap(k, p, firstr, lastr, vsizer)
        pivot = SQRT(pivot)

! --> Column k of Q is updated and copied into VCOL
!HPF$ ON HOME (firstq(k)), RESIDENT BEGIN
! --> This loop appears due to the ON HOME directive
        DO dum = ddly_LowBound(k), ddly_UpBound(k)
          aux => firstq(dum)%p
          DO i = 1, vsizeq(dum)
            ! ****** loop i body ******
          END DO
          size = vsizeq(dum)
        END DO
!HPF END ON
! --> VCOL is broadcast because it is replicated
        CALL ddly_aligncvs(vd_VCOL, md_Q, FirstDim)

! --> Dot-products between column k of Q and columns (k+1:n) of Q
        vsum(ddly_LowBound(k):ddly_UpBound(k)) = pivot;
        vsum(ddly_LowBound(k+1):ddly_UpBound(n)) = 0.0
!HPF$ INDEPENDENT, NEW (aux,i)
! --> Loop is partitioned
loopj1: DO j = ddly_LowBound(k+1), ddly_UpBound(n)
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
          ! ****** ON HOME body ******
!HPF$ END ON
        END DO loopj1            .
                                 .
                                 .
```

Figure 7.4: *Output of the HPF-2 compiler for the executable section of the MGS algorithm (first part)*

```
                          •
                          •
                          •

! --> Subrow (k,k:n) of R is updated
!HPF$ INDEPENDENT
! --> Loop is partitioned
        DO j = ddly_LowBound(k), ddly_UpBound(n)
          IF (ABS(vsum(j)) >= accuracy) THEN
            CALL append(k, vsum(j), firstr(j)%p, lastr(j)%p, vsizer(j))
          END IF
        END DO

! --> Submatrix of Q (columns k+1:n of Q) is updated
!HPF$ INDEPENDENT, NEW (aux,i,product)
! --> Loop is partitioned
loopj2: DO j = ddly_LowBound(k+1), ddly_UpBound(n)
!HPF$ ON HOME (firstq(j)), RESIDENT BEGIN
          ! ****** ON HOME body ******
!HPF$ END ON
        END DO loopj2

! --> Norms are updated
        norm(ddly_LowBound(k+1):ddly_UpBound(n)) = &
                 norm(ddly_LowBound(k+1):ddly_UpBound(n)) - &
                 vsum(ddly_LowBound(k+1):ddly_UpBound(n)) * &
                 vsum(ddly_LowBound(k+1):ddly_UpBound(n))

! --> Init x and vaux
        x = 0.0
!HPF$ INDEPENDENT
! --> Loop is partitioned
        DO j = ddly_LowBound(1), ddly_UpBound(n)
          vaux(j)%p => lastr(j)%p
        END DO

! --> Back-substitution
        DO i = rank, 1, -1
          acum = 0.0
!HPF$ INDEPENDENT, REDUCTION(acum)
! --> Loop is partitioned: local reduction
          DO j = ddly_LowBound(i+1), ddly_UpBound(rank)
            IF (ASSOCIATED(vaux(j)%p)) THEN
              IF (vaux(j)%p%index == i) THEN
                acum = acum + vaux(j)%p%value * x(j)
                vaux(j)%p => vaux(j)%p%prev
              END IF
            END IF
          END DO
! --> Global reduction
          CALL ddly_ReduceScalarSum(acum)

          DO dum = ddly_LowBound(i), ddly_LowBound(i)
            x(dum) = (qtb(dum)-acum)/vaux(dum)%p%value
            vaux(dum)%p => vaux(dum)%p%prev
          END DO
        END DO

      END DO main
```

Figure 7.4 (cont.): *Output of the HPF-2 compiler for the executable section of the MGS algorithm (last part)*

```
! --> The independent loops (DO j = globa, globb) are translated into
! --> (DO j = ddly_LowBound(globa), ddly_UpBound(globb))
! --> my_pe is a global variable which contains the processor identifier
! --> N$PES = # of PEs (global variable on the Cray T3D/T3E machine)

  INTEGER FUNCTION ddly_LowBound (i)
    INTEGER i
    ddly_LowBound = (i-1)/N$PES+1
    IF (my_pe < MOD(i-1,N$PES)) ddly_LowBound = ddly_LowBound+1
  END FUNCTION ddly_LowBound

  INTEGER FUNCTION ddly_UpBound (i)
    INTEGER i
    ddly_UpBound = i/N$PES
    IF (my_pe < MOD(i,N$PES)) ddly_UpBound = ddly_UpBound+1
  END FUNCTION ddly_UpBound
```

Figure 7.5: *DDLY functions for mapping loop iterations*

the array of processors (identified by the topology descriptor *td*), using the *BCS* scheme (consult Section 3.1). The change of storage scheme of matrix Q to an *LLCS* one (through the *ddly_ccs_to_llcs* routine) allows achieving the desired distribution (see the *LLCD* scheme of Figure 5.1). The implementation of this distribution scheme in two *DDLY* routines (instead of creating a new and complex one for our purposes) was to take advantage of the existing *DDLY* routine *ddly_bcs*, so that the new routine *ddly_ccs_to_llcs* is enough to have our distribution for linked lists (and there is no overhead using the combination of these two routines). The corresponding procedures for distributing matrix R were omitted because they follow the same structure as matrix Q (although, in this case, the effect is that the distributed structures are only initialized).

After the distribution, some data structures are aligned with the distributed matrices. As we can see in Figure 7.3, dense vectors are aligned with the first or the second dimension of matrix Q. Therefore, the HPF *ALIGN* directives are translated into the *DDLY* routines *ddly_alignv* with the appropriate parameters. The compressed vector *VCOL* is also aligned with the first dimension of matrix Q (that is, *VCOL* is replicated in all the processors) by means of the *ddly_aligncvs* routine.

A complete set of new initialization/distribution/alignment *DDLY* routines was developed to consider the wide variety of situations which can appear using the *SPARSE* directive.

7.2.2 Executable Section

Figure 7.4 displays the code generated by the extended HPF-2 compiler for the executable section of the MGS algorithm presented in Figures 6.11 and 6.16.

As we can see, the iterations of the loops annotated with the *INDEPENDENT*

directive are distributed on the array of processors by means of the *DDLY* functions *ddly_LowBound* and *ddly_UpBound*, applied to the lower and upper bounds of the loop, respectively (obviously, they are specific for the chosen data distribution). Figure 7.5 lists and explains both *DDLY* functions. These functions are also applied to the bounds of the operations performed in the HPF-2 code using subarray notation, because they are equivalent to independent *DO* loops (see, for instance, the updating of the norms). They were also used to translate global references to arrays into local references (as done, for example, in the norm swap).

The *ON HOME body* and *loop i body* comments of the code mean that the corresponding HPF-2 code of Figure 6.11 located in that place is directly copied (with no modification) to the output of the HPF-2 compiler.

The reduction operation performed in the column pivoting stage deserves special attention. It is split into two parts by the HPF-2 compiler (due to the aid of the *REDUCTION* clause):

- First, a local reduction (restricted to the condition of the IF construct) is carried out: each processor obtains the local absolute minimum of the entries of the distributed array *vsizeq* they own, as well as the corresponding index *p*.
- Second, a global reduction, using the *DDLY* function *ddly_ReduceLocMinAbs* is performed. It returns (in *p*) the global index of the entry of the distributed array *vsizeq* with minimum absolute value, using the information obtained in the local reduction.

A similar procedure is accomplished in the back-substitution stage for the variable *acum*. In this case, there is a local sum reduction of this variable and, next, a global reduction by means of the *ddly_ReduceScalarSum* procedure. Note that these are special reduction operations which involve IF constructs inside the reduction loop and this situation should be detected by a complex analysis phase of the compiler, as we pointed out in Subsection 6.5.3. Many *DDLY* routines to support a large variety of reduction operations have been implemented.

In the part of the code in which column *k* is updated and copied into the compressed vector *VCOL*, an auxiliary loop variable *dum* is created by the compiler so that the owner processor of the entry *firstq(k)* (see the corresponding *ON HOME* directive) is the only one that executes the piece of code inside this new *dum* loop. Note that references *firstq(k)* and *vsizeq(k)* of the original HPF-2 code in Figure 6.11 in this *ON HOME* body are substituted by the local references *firstq(dum)* and *vsizeq(dum)*, respectively. This approach with the auxiliary variable *dum* is also used at the end of the back-substitution stage.

Returning to the *VCOL* updating, the analysis stage of the compiler detects that the replicated compressed vector *VCOL* (this replication was indicated in the corresponding *ALIGN* directive) is updated in the *loop i body* comment. Therefore, once *VCOL* is updated by the owner processor of *firstq(k)*, it is broadcast

(their components *vcoli, vcolv* and *size* are broadcast) to all the processors, by means of the *ddly_aligncvs* routine. As we can see, latencies are saved by grouping together the information to be broadcast.

7.3 Experimental Results

Sparse MGS and Householder factorizations were used to experimentally test the corresponding parallel codes generated by our extended HPF-2 compiler, as we will show in the next subsections.

7.3.1 Parallel MGS Generated Code

We have executed the hypothetical output of the HPF-2 compiler for the sparse MGS algorithm (shown in Figures 7.3 and 7.4) on a Cray T3E multiprocessor, whose characteristics were described in Table 4.1 (consult [88] for further details about its architecture). It is important to remark that this is not an optimized hand-coded MGS program (as the one coded in C described in Section 3.2 and tested in Chapter 4), but it is the output of the compiler.

As we said in the previous section, we have extended the *DDLY* run-time library to support our HPF-2 extensions. The underlying routines we have used in the new *DDLY* functions to perform communications/synchronizations were the Cray SHMEM (Shared Memory access library) routines [11]. This library, specific for the Cray MPP line of computers (such as the Cray T3D and T3E) and available for Fortran and C interfaces, manages the Cray T3E following a NUMA shared memory model (consult Section 1.2). The main advantage of these routines is their very low latency, which leads to low execution times and efficient programs.

The choice of this library, which is difficult to program for a common user, for the run-time support was a weighed decision; in fact, the SHMEM routines are the communication primitives of the output generated by the *HPF_Craft* data-parallel compiler, mentioned in Section 6.2. Nevertheless, the main drawback of the SHMEM library is that its use is restricted to Cray computers. In order to make the output of the extended compiler portable to other machines, the communication core of the new *DDLY* functions was also implemented using MPI, at the expense of increasing the execution times with these slower routines.

Figures 7.6 and 7.7 present the corresponding execution times and speed-ups for several arrays of processors (1, 2, 4, 8 and 16 PEs), respectively. The test sparse matrices were taken from the Harwell-Boeing collection, and were described in Table 2.1. Harwell-Boeing matrix *WELL1850* (1850×712, 8758 nonzero elements, used in least squares problems in surveying) was also included in the experiments, as an example of the factorization of a rectangular matrix. As can be observed, the use of specific data distributions and data structures for sparse matrices embedded in the HPF-2 compiler leads to very good speed-ups.

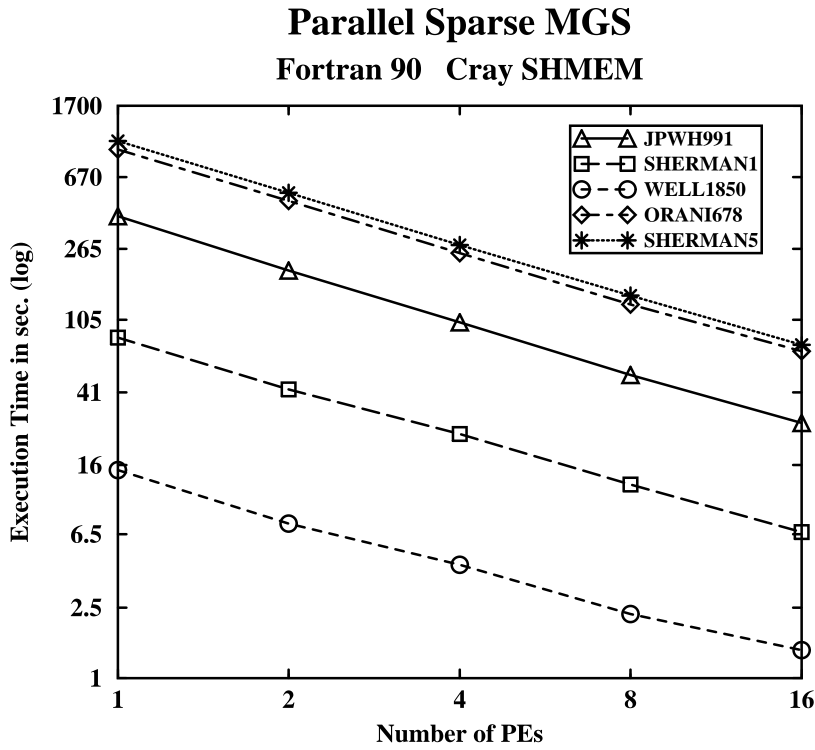

Figure 7.6: *MGS: execution times using F90 linked lists and Cray T3E SHMEM*

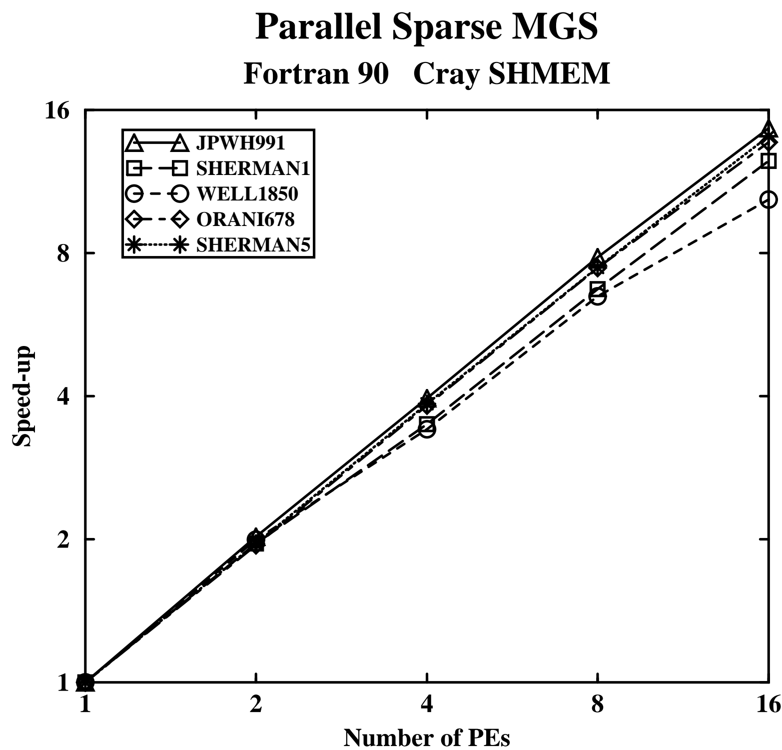

Figure 7.7: *MGS: speed-ups using F90 linked lists and Cray T3E SHMEM*

	Insertion	Deletion	Traversing	Total
Fortran 90	66.44s	53.57s	11.14s	129.15s
C	15.95s	10.59s	4.30s	30.84s

Table 7.1: *Benchmark for list management in Fortran 90 and C*

Superlinearity is even achieved for matrix *JPWH991* using two processors. This fact would be inconceivable using the classical dense distributions applied to sparse matrices.

Bigger matrices would report better results and would allow us to exploit parallelism using a large number of processors.

7.3.2 Parallel Householder Generated Code

Similar results are presented in Figures 7.8 and 7.9 for the Householder algorithm. Note that the same scale is used for the graphs which represent the execution times of MGS and Householder, in order to compare them properly. The speed-ups obtained for Householder are better than the ones for MGS; we achieve superlinearity for more cases, even for eight processors (matrix *SHERMAN5*).

A curious matter, for both Fortran 90 MGS and Householder algorithms, is that their execution times are very high compared with the corresponding manual C algorithms which use doubly-linked lists on the Cray T3D (see Tables 4.5 and 4.6), taking also into account that the Fortran 90 algorithms are executed on DEC Alpha processors at 300 Mhz (Cray T3E), whereas the C algorithms were executed on slower Alpha processors at 150 Mhz (Cray T3D). This fact should lead, in theory, to lower execution times for the Fortran 90 algorithms.

In order to explain this strange situation, we have designed a simple benchmark to compare the list management in Fortran 90 and C on a single Alpha processor at 300 Mhz. This benchmark consisted in performing repeatedly a great number of insertion, deletion and traversing operations in linked lists. The execution times are shown in Table 7.1 and it clearly explains the high running times of the Fortran 90 algorithms. As we can see, for this particular benchmark, the total execution time in Fortran 90 is more than four times slower than in C. One possible reason of this fact is that Fortran 90 is not as flexible as C and this implies many type and memory checkings at run-time for dynamic memory operations, which results in an increase of the execution times. Fortran 90 compilers (at least for the Cray T3E) should be greatly improved and optimized in this aspect of pointers/dynamic memory to be competitive with C compilers.

As a consequence of the high execution times of the Fortran 90 algorithms and the use of the fast SHMEM routines, the speed-ups are obviously much better than the ones of the corresponding C algorithms coded by hand, as we showed experimentally in Figures 7.7 and 7.9.

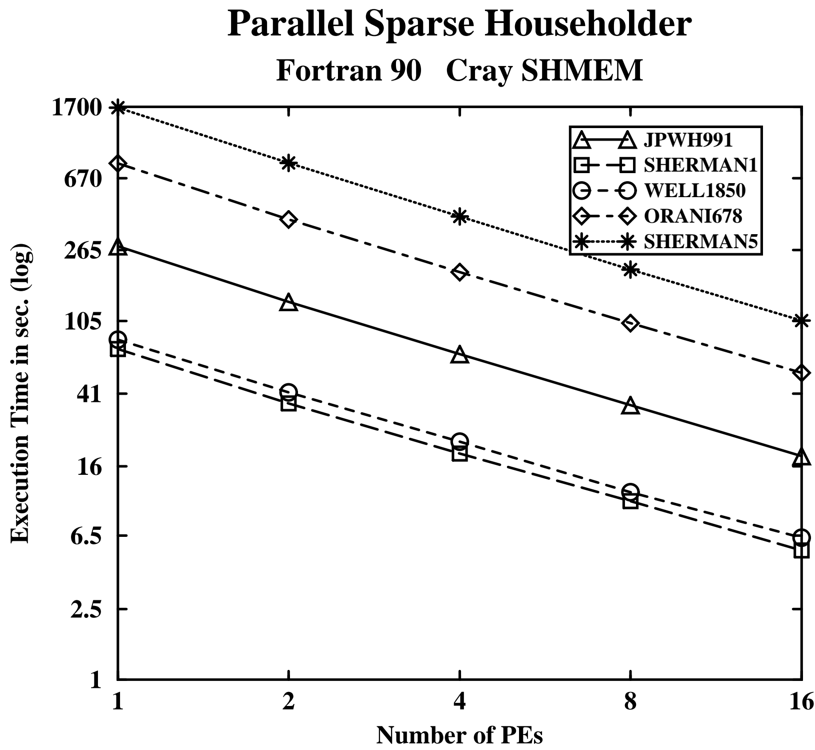

Figure 7.8: *Householder: execution times using F90 linked lists and Cray T3E SHMEM*

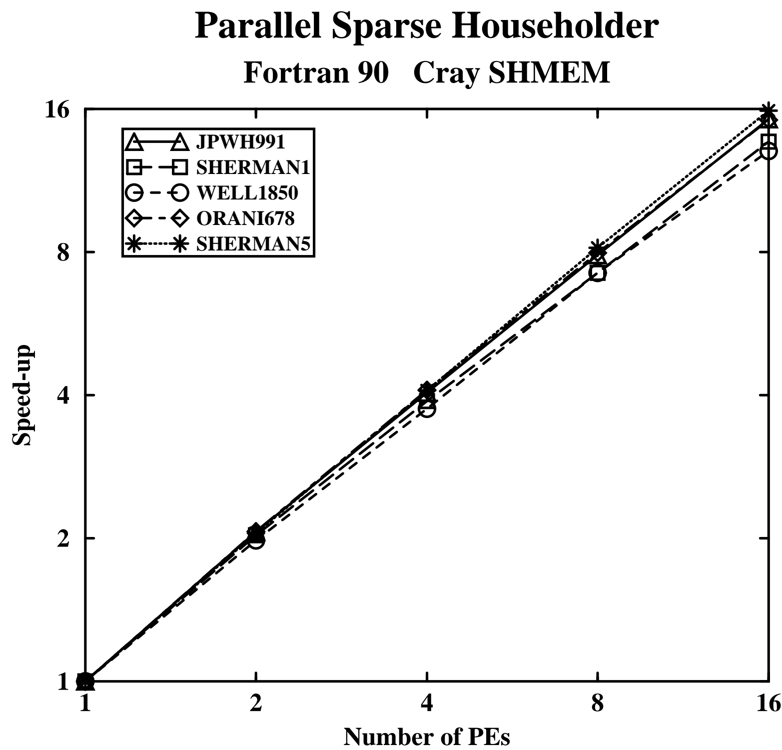

Figure 7.9: *Householder: speed-ups using F90 linked lists and Cray T3E SHMEM*

Matrix	CHAOS LLCS	Manual LLCS	Ratio
SHL400	52s	6s	8.66
JPWH991	615s	59s	10.42
SHERMAN1	243s	27s	9.00
MAHINDAS	179s	21s	8.52
ORANI678	1228s	134s	9.16
SHERMAN5	1685s	191s	8.82

Table 7.2: *Execution times for the parallel sparse MGS using manual LLCS and the CHAOS implementation*

7.4 Comparison with the CHAOS Approach

In order to compare our parallel solutions with others based mainly on run-time techniques, we have additionally implemented the sparse MGS algorithm using the CHAOS run-time library [79][86]. This library uses the inspector-executor paradigm to deal with irregular data accesses.

Routines from the CHAOS library were inserted to rebalance the load (and data) of the parallel sparse MGS algorithm. Basically, through these routines, we have generated a translation table which assigns the global indices of matrix A to the different processors following an irregular model (specifically, a cyclic LLCS distribution). This table is distributed across the processors and used by the *localize* CHAOS routine to translate the global indices into local indices within each processor. It also generates a communication schedule used to gather the off-processor data needed during computation, and to scatter back local copies after computation.

Table 7.2 shows a comparison of the execution times using the cyclic LLCS pseudo-regular data distribution (see Section 6.4) in a manual MGS algorithm (this manual algorithm is the same as the one described in Section 3.2), as opposed to the same distribution but implemented with CHAOS routines (all codes were written in C). The execution times have been taken in a cluster of 16 Sun SPARCstation 4 with 85-MHz microSPARC-II processors in a PVM message-passing environment. The test sparse matrices were taken from the Harwell-Boeing suite (described in Table 2.1). The entry *ratio* represents the quotient T_{CHAOS}/T_{Manual} and, as we can see, for all the tested matrices, the cyclic LLCS pseudo-regular approach (which is the one proposed in our HPF-2 extensions of Section 6.4) obtains almost one order of magnitude of improvement in the execution times. For other sparse problems without pivoting and fill-in (and using BRS/BCS distributions), similar results are presented in [102]. Therefore, pseudo-regular data distributions obtain better performance than the CHAOS general inspector-executor model.

The CHAOS approach has a large number of communications and high memory overhead, as a consequence of accessing a large distributed data addressing table. This results in high execution times. On the other hand, as we also experi-

mentally proved in previous chapters, the cyclic LLCS pseudo-regular distribution is adequate for sparse matrix problems (in particular, QR algorithms) because it exploits data and computation locality and minimizes communications. It does not require additional storage or communications for addressing non-local data, as all the processors know where the data are located using simple operations.

7.5 Towards Automatic Parallelization

The limited skill of compilers to detect parallelism in codes is an important drawback to the use of supercomputers because it compels programmers to tediously parallelize their programs by hand, as we did in the first part of this thesis. This fact adds a significant level of complexity to the programming task. Because of these compiler limitations, conventional programming languages are usually extended with directives to provide the compiler with the information that it cannot obtain by itself, as discussed in Chapter 6 for HPF.

Nowadays, many efforts are being made to improve compilers with complex techniques to parallelize programs automatically. This way, the acceptance and use of multiprocessors would be enhanced due to this transparency to programmers. Many of these techniques to generate optimized parallel codes are described in [106][111]. The SUIF [50] and Polaris [16] parallelizing compilers are two representative examples of current active research on automatic program parallelization.

7.5.1 The SUIF and Polaris Compilers

The SUIF (Stanford University Intermediate Format) compiler, developed by the Stanford Compiler Group, is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. SUIF transforms Fortran 77 and C sequential programs into SPMD code for shared address space machines. This parallel code includes calls to a portable run-time library, which is currently available for SGI machines, the Stanford DASH multiprocessor and Digital AlphaServers.

The SUIF system consists of a set of compiler passes (implemented as separate programs), each one of them performs a single analysis or transformation of the code. These passes work together by using a common intermediate format to represent programs. New passes can be freely inserted at any point in a compilation. These facts make SUIF easy to modify and extend, becoming a very adequate environment for evaluating new compiler techniques. As we can see, although this approach is somewhat inefficient, it is very flexible.

The passes that SUIF runs for parallelization are: constant propagation, scalarizing array accesses, forward propagation, normalization, induction variable detection, constant folding, scalar privatization analysis, reduction recognition, dependence preprocessing, parallelism and locality optimization, and parallel code

```
DO j = 1, n
  vsum(j) = 0.0
  DO i = col(j), col(j+1)-1
    vsum(j) = vsum(j) + data(i) * vcol(row(i))
  END DO
END DO
```

Figure 7.10: *Sparse matrix-dense vector product*

generation. After running the parallelization passes, SUIF can generate C code or MIPS processor code.

The Polaris compiler includes optimization techniques (automatic detection of parallelism and data distribution) to translate a Fortran 77 sequential program into an output (Fortran 77 plus parallel directives) which can be executed efficiently on a parallel computer: shared memory multiprocessors (such as an SGI Challenge machine) and scalable machines with a global address space (such as the Cray T3D). Currently, a Polaris version for a network of computers is being developed.

The Polaris restructurer also performs its transformations in several compilation passes. Polaris includes several analysis techniques based on both static and dynamic (run-time) parallelization strategies, which rest on a careful research on the characteristics of real Fortran codes. These techniques, such as data dependence analysis (at compile-time or dynamically at run-time), symbolic program analysis, induction variable recognition, interprocedural analysis, array privatization, reduction variable recognition, etc. are widely described and tested in [15] and [17].

7.5.2 Automatic Parallelization of Sparse Computations

In this thesis, we have focused on sparse matrix operations, which involve irregular memory access patterns. As an example of this kind of operations, the sparse matrix-dense vector product is depicted in Figure 7.10. This product is a common operation performed in our QR algorithms; see, for instance, expression (2.7) in Figure 2.1, which represents a dense matrix-vector product (the vector is the column k of matrix A). In our Fortran 77 code, this operation is converted into a sparse matrix-dense vector product. The sparse matrix is stored in a CCS format using three vectors (*data, row, col*) and column k of the matrix is stored in vector *vcol*; the dense array *vsum* stores the result of this product. The arrays *vcol* and *vsum* were described in the parallel MGS C algorithm (Section 3.2), but note that in our Fortran 77 code, *vcol* is a dense array, instead of a packed vector.

The sparse matrix-dense vector product is also used in the Householder algorithm to compute vector w (see expression (2.12) of Figure 2.4). The dense vector of the product in our Fortran 77 algorithm is the Householder vector v (in the

```
static void _MAIN___0_func(int _my_id)
{
 _my_struct_ptr = _suif_aligned_args;
 j = _my_struct_ptr->_field_0;
 vsum = _my_struct_ptr->_field_1;
 data = _my_struct_ptr->_field_2;
 vcol = _my_struct_ptr->_field_3;
 row = _my_struct_ptr->_field_4;
 col = _my_struct_ptr->_field_5;
 _my_nprocs = *_suif_aligned_my_nprocs;
 for (__priv_j0 = max(n * _my_id / _my_nprocs + 1, 1);
      __priv_j0 < min(n * (_my_id + 1) / _my_nprocs + 1, n+1); __priv_j0++)
   {
    suif_tmp3 = &((double *)vsum)[__priv_j0 - 1];
    suif_tmp4 = suif_tmp3;
    suif_tmp5 = suif_tmp4;
    suif_tmp6 = suif_tmp5;
    *suif_tmp6 = 0.0;
    __s2c_tmp7 = ((int *)col)[(long)(__priv_j0 + 0)] - 1;
    for (i = ((int *)col)[__priv_j0 - 1]; i <= __s2c_tmp7; i++)
      {
       *suif_tmp5 = *suif_tmp6 + ((double *)data)[i - 1] *
                    ((double *)vcol)[((int *)row)[i - 1] - 1];
      }
   }
 if (_my_id == _my_nprocs - 1)
   {
    *j = __priv_j0;
   }
 return;
}

extern int MAIN__()
{
 _doall_level_result = suif_doall_level();
 if (0 < _doall_level_result)
   {
    for (j = 1; j <= n; j++)
      {
       suif_tmp1 = &vsum[j - 1];
       *suif_tmp1 = 0.0;
       __s2c_tmp2 = col[j - 1];
       __s2c_tmp3 = col[(long)(j + 0)] - 1;
       if (__s2c_tmp2 <= __s2c_tmp3)
         {
          suif_tmp0 = suif_tmp1;
          for (i = __s2c_tmp2; i <= __s2c_tmp3; i++)
            {
             *suif_tmp0 = vsum[j - 1] + data[i - 1] * vcol[row[i - 1] - 1];
            }
         }
      }
   }
 else
   {
    struct _BLDR_struct_000 *_MAIN___0_struct_ptr;
    suif_start_packing(_MAIN___0_func, "_MAIN___0_func");
    *_suif_aligned_task_f = _MAIN___0_func;
    _MAIN___0_struct_ptr = _suif_aligned_args;
    suif_named_doall(_MAIN___0_func, "_MAIN___0_func", 0, n, 1);
   }
}
```

Figure 7.11: *Summarized output of the SUIF compiler for the sparse matrix-dense vector product*

```
C$DOACROSS LOCAL(I,J),SHARE(N,VSUM,COL,DATA,VCOL,ROW),
C$& MP_SCHEDTYPE=INTERLEAVED,CHUNK=2

CSRD$ LOOPLABEL 'PROD_do#1'
      DO j = 1, n, 1
        vsum(j) = 0.0
        i = col(j)
CSRD$ LOOPLABEL 'PROD_do#1#1'
        DO i = 1, col(1+j)+(-col(j)), 1
          vsum(j) = vsum(j)+data((-1)+col(j)+i)*vcol(row((-1)+col(j)+i))
        END DO
      END DO
```

Figure 7.12: *Summarized output of the Polaris compiler for the sparse matrix-dense vector product*

sparse C algorithm v is a packed vector).

Figures 7.11 and 7.12 show the core of the output of the SUIF and Polaris compilers for the code of Figure 7.10, respectively. As we indicated in the earlier subsection, the SUIF output is SPMD C code (we have omitted many lines of code to simplify) and the Polaris restructurer output is composed of Fortran 77 code plus a set of compiler directives (Polaris directives begin with the keyword *CSRD$*).

The SUIF and Polaris compilers parallelize the outer loop j of the sparse matrix-dense vector code of Figure 7.10; but, unfortunately, there are many other irregular access patterns in our sparse QR codes that greatly complicate the automatic detection of parallelism and, in many situations, following a conservative approach, potential parallel programs are not parallelized, which decreases the speed-ups of the output parallel codes.

Asenjo et al. propose in [5] automatic parallelization techniques to detect and exploit parallelism on a collection of real codes with irregular access patterns (sparse factorizations, molecular dynamics and fluid dynamics computations, etc.), and evaluate the effectiveness of these methods with the aim of being included in the Polaris restructurer. We have applied two of these techniques to the Fortran 77 code of the MGS and Householder QR decomposition algorithms. These techniques are: the test of monotonicity of index arrays and the test of non-overlapping ranges of induction variables, and we describe them in the next two subsections.

The reason of using Fortran 77 QR algorithms is that the basis of Polaris is Fortran 77 and we cannot use Fortran 90 capabilities, as we have done in our HPF data-parallel approach. Therefore, pointer data types and dynamic data structures cannot be analyzed by the Polaris compiler. These sequential Fortran 77 QR codes are simplified and follow the same style as the Craft MGS algorithm described in Section 6.2.

However, there is active research and increasing interest in dependence ana-

```
DO j = 1, n
  DO i = col(j), col(j+1)-1
    data(i) = ...
  END DO
END DO
```

Figure 7.13: *Subscripted array subscripts in the QR codes*

lysis of pointer references to be included in a parallelizing compiler, which is certainly not a trivial task. Matsumoto et al. describe in [68] a new method to analyze dependences between pointer references in Pascal and Fortran 90 languages, which mainly appear in the management of dynamic data structures, such as linked lists or trees. Amme and Zehendner [3] present an efficient data dependence analysis method in programs with pointers based on solving a data flow problem.

7.5.3 Testing Monotonicity of Index Arrays

The appearance of subscripted subscripts in the loops of the code usually forces data dependence analyzers to conclude conservatively that those loops are not parallelizable.

These subscripted array subscripts are very common in sparse codes and are difficult to handle by the automatic parallelizers. One common pattern of our QR algorithms involves the use of subscripted array subscripts in loop bounds, as shown in Figure 7.13, in which an array variable that traverses that loop is written.

The condition for the outermost loop j of this code to be parallelized (*doall* loop) is given by the following expression:

$$\bigcap_{j=1}^{n} [col(j), col(j+1) - 1] = \emptyset \tag{7.1}$$

that is, $\forall j$, the range [col(j), col(j+1)-1] must be non-overlapping.

Although this condition may not hold for other patterns with subscripted array subscripts, for our sparse codes the index arrays are inherently monotonic. This is because the matrices in our sparse codes are represented using a compressed format (a column-wise storage scheme, CCS, in our example). Each entry j of array *col* stores the position in *data* where column j of the sparse matrix begins and, therefore, the index array *col* is non-decreasing.

The analysis of this kind of patterns is difficult to accomplish at compile-time. However a combination of compile-time analysis and run-time tests would allow to prove that these index arrays are non-decreasing. Sometimes, the index array is read from input, as occurs with our QR code, in which the coefficient matrix

```
DO j = 1, n
  col2(j) = col1(j)
END DO

DO k = 1, n
  bufindex = MOD(k,2)*shift + 1
  DO j = k+1, n
    cptr1 = bufindex
    DO i = col1(j), col2(j)
      data(bufindex) = data(i)
      bufindex = bufindex + 1
    END DO
    cptr2 = bufindex - 1
    IF (fill_in_cond) THEN
      cptr2 = cptr2 + 1
      data(bufindex:bufindex+inc) = ...
      bufindex = bufindex + inc
    END IF
    DO i = col2(j)+1, col1(j+1)-1
      data(bufindex) = data(i)
      bufindex = bufindex + 1
    END DO
    col1(j) = cptr1
    col2(j) = cptr2
  END DO
  col(n+1) = bufindex
END DO
```

Figure 7.14: *Fill-in stage of the Fortran 77 sparse QR algorithms*

is read from a file in a CCS format. In this case, proving that the ranges do not overlap is equivalent to testing that $\forall j, 1 \leq j \leq n, col(j+1) \geq col(j)$. This test, whose running time overhead is negligible, should be included, when possible, as part of the initial input stage. As the test is basically a reduction operation across the array *col*, it can also be executed in parallel to decrease the test execution time.

7.5.4 Testing Non-Overlapping Ranges of Induction Variables

In order to describe this test, Figure 7.14 shows a simplified code of the process of updating coefficient matrix A in QR algorithms, the fill-in stage, about which we will not go into details.

The outermost loop k is executed for the n columns of the sparse matrix A, stored in CCS format (*data, row, col*), as we said in the previous subsection; *col1, col2* are auxiliary index arrays. This piece of code includes array references to the entries of the sparse matrix (stored in array *data*) via a linear induction variable *bufindex*, which is conditionally incremented (see the IF block of Figure 7.14). In order to prove the independence (non-overlap) of ranges of array references through an induction variable, a static analysis may be used to reveal non-overlapping conditions which can be tested at run-time.

Let us focus on the example code of Figure 7.14. The two sufficient conditions the $j(k+1:n)$ loop must fulfill to be executed in parallel are:

- First, the range of *bufindex* must not overlap the same range of *bufindex* for iterations executed on other processors; that is, output dependences are not present.
- Second, the range of *bufindex* in this loop must not overlap the range [col1(j), col1(j+1)-1], as shown in the following expression:

  $$\forall j,\ k+1 \leq j \leq n,\ range(bufindex) \bigcap [col1(j), col1(j+1)-1] = \emptyset \quad (7.2)$$

  This is equivalent to asserting that there are no flow or anti-dependences.

In order to prove these two conditions, it must first be determined that index arrays *col1* and *col2* are non-decreasing. As we can see in Figure 7.14, these arrays are reassigned in each iteration of loop $j(k+1:n)$, which makes the analysis of the array access pattern difficult. Nevertheless, simple conditions under which these index arrays will be non-decreasing can be determined statically at compile-time, as we will next show.

The induction variable *bufindex* is never decremented in loop j. Besides, it is conditionally increased under the condition *fill_in_cond* in a positive quantity (*inc*). Moreover, the initial assignment of the first $j(1:n)$ loop of Figure 7.14 assures that the first $i(col1(j):col2(j))$ loop is executed at least one iteration; therefore, one of the invariant conditions of this loop is that the induction variable *bufindex* is strictly increasing. If arrays *col1* and *col2* are initially non-decreasing, this invariant condition is, by induction, a sufficient condition to assert that they will remain non-decreasing during the execution of the outermost loop k. As a conclusion, the proof of the non-decreasing nature of these index arrays has been reduced to executing the test: $\forall j, 1 \leq j \leq n, col(j+1) \geq col(j)$, once at run-time.

As a result of the fact that *bufindex* is strictly increasing in the loop $j(k+1:n)$, we have also proven that there are no output dependences across the iterations of this loop (first condition).

The next step is to prove the second condition (see expression (7.2)), using a simple test which compares the upper bound of *bufindex* to the lower bound of *col1* and the lower bound of *bufindex* to the upper bound of *col1*. The presence of a conditional increment in *bufindex* (by means of *fill_in_cond*) complicates the analysis; however, we can use an estimate of the maximum value of *bufindex* by determining an upper bound across the entire iteration space of the $j(k+1:n)$ loop. Given that *col1* and *col2* are non-decreasing and the strictly increasing nature of *bufindex*, then $col2(j) \geq col1(j)$; thus we know the tripcount of the $i(col1(j):col2(j))$ loop and we can conservatively assume that the *fill_in_cond* condition is always fulfilled.

Together these facts lead to the run-time test of Figure 7.15 to prove the non-overlapping nature of reads and writes to array *data*. That is, the test concludes the proof that the iterations of the $j(k+1:n)$ loop are independent.

```
min_i = col(k+1)
max_i = col(n+1)-1
min_bufindex = bufindex
max_bufindex = bufindex + (max_i-min_i) + ((n-k)*inc)

IF ((min_bufindex .GT. max_i) .OR. (max_bufindex .LT. min_i) THEN
  parallel_ex = .TRUE.
ELSE
  parallel_ex = .FALSE.
END IF
```

Figure 7.15: *Test of non-overlapping ranges of the variable bufindex*

This test (which has little overhead) must be placed outside the $j(k+1:n)$ loop. When it is true, this loop may be executed in parallel; otherwise, it must conservatively be executed serially.

Conclusions

This thesis has covered a wide variety of subjects in the field of parallel sparse numerical computations: from parallel sparse algorithms constructed by hand up to automatic parallelization techniques of sparse codes.

Thus, we have used different numerical libraries for sparse and dense matrix algorithms, parallel distributed memory target machines (Fujitsu AP1000, Cray T3D, Cray T3E, a network of workstations), message-passing libraries (PVM, MPI, Cray SHMEM library, AP1000 CellOS routines), run-time libraries (DDLY, CHAOS), compiler construction tools (Cocktail), data-parallel compilers (HPF, Craft), automatic parallelizing compilers (SUIF, Polaris), etc.

The main contributions of the thesis can be summarized in the following eight items:

- The implementation of portable parallel sparse rank-revealing QR algorithms on MIMD distributed memory supercomputers, with application to solving least squares problems, which are often used in many active scientific areas. The choice of the data distribution and the dynamic data structures was essential to obtain efficient parallel algorithms, as we showed experimentally in the corresponding chapters.

- The inclusion of a general heuristic strategy, valid for the three QR methods (Modified Gram-Schmidt, Householder reflections and Givens rotations) and for any pattern of the matrix to be factorized, in order to preserve the sparsity rate of the coefficient matrix during factorization. This strategy obtains acceptable reductions of fill-in and preserves the numerical stability of the computations. Using this fill-in reduction criterion, based on column pivoting and with minimum execution time overhead, the sparse QR direct methods can be competitive, in some applications, with iterative algorithms.

- A local pivoting strategy was obtained to reduce the number of communications (message-passing) when applying the column pivoting associated with the fill-in reduction criterion. Thus, the execution times of the parallel QR algorithms are reduced and the efficiencies are increased. This local pivoting can be easily adapted to any other factorization algorithms (LU, Cholesky $\cdots$) that employ pivoting.

- The development of *3LM*, a portable parallel library for sparse matrix computations involving pivoting and fill-in operations, in order to facilitate the programming of sparse algorithms on distributed memory multiprocessors. This library, called by means of a simple interface, includes a wide variety of features for sparse matrices and vectors, such as dynamic data structures (mainly based on linked lists), data distributions/redistributions, data replication operations, reduction operations, routines to manage pivoting and fill-in, list and vector operations, etc.

- A syntax proposal to extend the standard High Performance Fortran data-parallel language with new features to support sparse matrix operations. These features, focused on the new *!HPF$ SPARSE* compiler directive, include data distributions/alignments and data structures (based on Fortran 90 derived data types) specific for sparse matrices and fill-in operations. Thereby, the efficiencies of the sparse data-parallel algorithms are greatly improved.

- A compiler module to support the syntax proposal of the previous point. This module consists in the developement of a source code analyzer for extended HPF (lexical and syntactic analysis) and the description of the code parallelizator and generator, following an SPMD paradigm. The module was designed to be embedded and integrated in an HPF compiler and to easily include new extensions to the HPF language.

- A run-time library for the execution of the parallel tasks of the SPMD code generated by the compiler module. It consists in a complete and portable set of routines to perform data distributions and alignments, reductions, array index translations, etc.

- An introduction to the analysis and application of automatic parallelization techniques (monotonicity of index arrays and non-overlapping ranges of induction variables) suitable for sparse matrix operations, focusing on our sparse QR codes.

As future research work, we will consider three main goals (related exclusively to the second part of the thesis) to complement the task developed in this dissertation:

- First, we intend to complete and improve the compiler module and the run-time library of the HPF extension for sparse computations, in order to generate an optimized SPMD code.

- Second, we will study the feasibility of including new features in the HPF compiler for another set of numerical scientific irregular problems provided that this inclusion implies minimum changes in the compiler. Consequently, new data structures and data distributions should be analyzed to improve the efficiencies of these irregular applications.

- The third research line is to go deeply into new automatic parallelization techniques for codes with irregular pattern accesses (focusing on pointers and dynamic data structures) liable to be included in a code restructurer for parallel compilers.

Bibliography

[1] AEA Technology, Harwell Laboratory. *Harwell Subroutine Library. Specifications*, Release 12, Jan 1996.

[2] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1985.

[3] W. Amme and E. Zehendner. Data Dependence Analysis in Programs with Pointers. In *Sixth Workshop on Compilers for Parallel Computers, CPC'96*, Special Issue of the Volume Konferenzen des Forschungszentrums Jülich, Vol.21, pages 371-382. Aachen, Germany, Dec 1996.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. *LAPACK User's Guide*, second edition. SIAM Pub., Philadelphia, 1995.

[5] R. Asenjo, E. Gutiérrez, Y. Lin, D. Padua, B. Pottenger and E.L. Zapata. *On the Automatic Parallelization of Sparse and Irregular Fortran Codes.* Technical Report 1512, CSRD, University of Illinois at Urbana-Champaign, Dec 1996.

[6] R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata. Sparse Block and Cyclic Data Distributions for Matrix Computations. In *High Performance Computing: Technology, Methods and Applications*, Elsevier Science B.V., pages 359-377. The Netherlands, 1995.

[7] R. Asenjo, J. Touriño, R. Doallo, O. Plata and E.L. Zapata. HPF-2 Support for Sparse Computations Involving Pivoting and Fill-in. Submitted to *IEEE Transactions on Parallel and Distributed Systems.*

[8] R. Asenjo and E.L. Zapata. Sparse LU Factorization on the Cray T3D. In *Int'l Conference on High-Performance Computing and Networking, HPCN'95*, Springer-Verlag LNCS 919, pages 690-696. Milan, Italy, 1995.

[9] G. Bandera, G.P. Trabado and E.L. Zapata. Extending Data-Parallel Languages for Irregularly Structured Applications. In *NATO Adv. Res. Workshop on High Performance Computing: Technology and Applications.* Cetraro, Italy, 1996.

[10] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Pub., 1994.

[11] R. Barriuso and A. Knies. *SHMEM User's Guide for Fortran*, Revision 2.2, Cray Research, Inc., 1994.

[12] C. Bendtsen, P.C. Hansen, K. Madsen, H.B. Nielsen and M. Pinar. Implementation of QR Up and Downdating on a Massively Parallel Computer. *Parallel Computing*, 21:49-61, 1995.

[13] A.J.C. Bik. *Compiler Support for Sparse Matrix Computations.* Ph.D. Dissertation, University of Leiden, The Netherlands, 1996.

[14] C.H. Bischof. Adaptive Blocking in the QR Factorization. *The Journal of Supercomputing*, 3:193-208, 1989.

[15] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu. *Advanced Program Restructuring for High-Performance Computers with Polaris.* Technical Report 1473, CSRD, University of Illinois at Urbana-Champaign, Jan 1996.

[16] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78-82, Dec 1996.

[17] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger and P. Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37-47, 1994.

[18] P. Brezany, K. Sanjari, O. Cheron and E. van Konijnenburg. Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler. In *Int'l Conference on High-Performance Computing and Networking, HPCN'95*, Springer-Verlag LNCS 919, pages 526-531. Milan, Italy, 1995.

[19] J.C. Cabaleiro and T.F. Pena. Solving Sparse Triangular Systems on the AP1000 Multicomputer. In *Sixth Parallel Computing Workshop, PCW'96*, Fujitsu Laboratories Ltd., pages S-D-1 to S-D-7. Kawasaki, Japan, Nov 1996.

[20] S. Chandrasekaran and I.C.F. Ipsen. Analysis of a QR Algorithm for Computing Singular Values. *SIAM J. Matrix Anal. Appl.*, 16(2):520-535, Apr 1995.

[21] B. Chapman, P. Mehrotra and H. Zima. HPF+: A New Language and Implementation Mechanisms for the Support of Advanced Irregular Applications. In *Sixth Workshop on Compilers for Parallel Computers, CPC'96*, Special Issue of the Volume Konferenzen des Forschungszentrums Jülich, Vol.21, pages 195-206. Aachen, Germany, Dec 1996.

[22] B. Chapman, P. Mehrotra and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1:31-50, 1992.

[23] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker and R.C. Whaley. *ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers -Design Issues and Performance.* Technical Report CS-95-283, Dep. Computer Science, University of Tennessee, Knoxville, 1995.

[24] J. Chun, T. Kailath and H. Lev-Ari. Fast Parallel Algorithms for QR and Triangular Factorization. *SIAM J. Sci. Stat. Computing*, 8(6):899-913, Nov 1987.

[25] Cray Research, Inc. *Cray T3D. Technical Summary*, Sep 1993.

[26] P. Crooks and R.H. Perrott. Language Constructs for Data Partitioning and Distribution. *Scientific Programming*, 4:59-85, 1995.

[27] D. Culler, J.P. Singh and A. Gupta. *Parallel Computer Architecture.* Morgan Kaufmann Publishers, alpha draft available at http://http.cs.berkeley.edu/~culler, 1997.

[28] R. Doallo, B.B. Fraguela, J. Touriño and E.L. Zapata. Parallel Sparse Modified Gram-Schmidt QR Decomposition. In *Int'l Conference on High-Performance Computing and Networking, HPCN'96*, Springer-Verlag LNCS 1067, pages 646-653. Brussels, Belgium, Apr 1996.

[29] R. Doallo, J. Touriño and F.M. Hermo. Sparse Matrix Operations in Vector and Parallel Processors. *International Series on Advances in High Performance Computing, Volume 3: Applications of High Performance Computing in Engineering V*, pages 43-52. Computational Mechanics Publications, 1997.

[30] R. Doallo, J. Touriño and F.F. Rivera. Solving the Least Squares Problem for Sparse Matrices on the AP1000. In *Fourth Parallel Computing Workshop, PCW'95*, Fujitsu Laboratories Ltd., pages 101-111. London, UK, Sep 1995.

[31] R. Doallo, J. Touriño and E.L. Zapata. Sparse Householder QR Factorization on a Mesh. In *Fourth Euromicro Workshop on Parallel and Distributed Processing, PDP'96*, IEEE Computer Society Press, pages 33-39. Braga, Portugal, Jan 1996.

[32] J. Dongarra, J.R. Bunch, C.B. Moler and G.W. Stewart. *LINPACK User's Guide*, second edition. SIAM Pub., Philadelphia, 1979.

[33] I.S. Duff, A.M. Erisman and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, 1986.

[34] I.S. Duff, R.G. Grimes and J.G. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection.* Technical Report TR-PA-92-96, CERFACS, Oct 1992.

[35] T.M.R. Ellis, I.R. Philips and T.M. Lahey. *Fortran 90 Programming.* Addison-Wesley Publishing Company Inc., 1994.

[36] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng and M. Wu. *Fortran D Language Specification.* Technical Report COMP TR90-141, Dep. Computer Science, Rice University, 1990.

[37] B.B. Fraguela, J. Touriño and R. Doallo. Sparse Orthogonalization on the Cray T3D. In *Second European Cray MPP Workshop.* Edinburgh, UK, Jul 1996.

[38] Fujitsu Laboratories Ltd. *AP1000 Program Development Guide*, third edition, Jul 1993.

[39] Fujitsu Laboratories Ltd. *CASIM User's Guide*, fourth edition, Aug 1991.

[40] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V.S. Sunderam. *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[41] J.A. George, M.T. Heath and E.G.Y. Ng. A Comparison of some Methods for Solving Sparse Linear Least Squares Problems. *SIAM J. Sci. Stat. Computing*, 4:177-187, 1983.

[42] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, second edition, 1989.

[43] D. Gries. *The Science of Programming.* Springer-Verlag New York Inc, 1985.

[44] J. Grosch. *Cocktail: Toolbox for Compiler Construction. Lark - An LR(1) Parser Generator with Backtracking.* Technical Report 32, University of Karlsruhe, Germany, Aug 1996.

[45] J. Grosch. *Cocktail: Toolbox for Compiler Construction. Rex - A Scanner Generator.* Technical Report 5, University of Karlsruhe, Germany, Apr 1996.

[46] J. Grosch. *Cocktail: Toolbox for Compiler Construction. Toolbox Introduction.* Technical Report 25, University of Karlsruhe, Germany, Nov 1994.

[47] J. Grosch and B. Vielsack. *Cocktail: Toolbox for Compiler Construction. The Parser Generators Lalr and Ell.* Technical Report 8, University of Karlsruhe, Germany, Jul 1992.

[48] M. Gulliksson. Backward Error Analysis for the Constrained and Weighted Linear Least Squares Problem when Using the Weighted QR Factorization. *SIAM J. Matrix Anal. Appl.*, 16(2):675-687, Apr 1995.

[49] J.B. Haag and D.S. Watkins. QR-Like Algorithms for the Nonsymmetric Eigenvalue Problem. *ACM Transactions on Mathematical Software*, 19(3):407-418, Sep 1993.

[50] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion and M.S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84-89, Dec 1996.

[51] D.R. Hare, C.R. Johnson, D.D. Olesky and P. van den Driessche. Sparsity Analysis of the QR Factorization. *SIAM J. Matrix Anal. Appl.*, 14(3):655-669, Jul 1993.

[52] M.T. Heath. *Scientific Computing. An Introductory Survey.* McGraw-Hill, 1997.

[53] B. Hendrickson. Parallel QR Factorization Using the Torus-Wrap Mapping. *Parallel Computing*, 19:1259-1271, 1993.

[54] J.L. Hennessy and D.A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, second edition, 1996.

[55] T. Horie and M. Ikesaka. AP1000 Software Environment for Parallel Programming. *Fujitsu Sci. Tech. Journal*, 29(1):25-31, Mar 1993.

[56] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0. *Scientific Programming*, 2(1-2):1-170, 1993.

[57] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0, Oct 1996.

[58] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato and M. Ikesaka. Third Generation Message Passing Computer AP1000. In *Int'l Symposium on Supercomputing*, pages 46-55, Nov 1991.

[59] H. Ishihata, T. Horie and T. Shimizu. Architecture for the AP1000 Highly Parallel Computer. *Fujitsu Sci. Tech. Journal*, 29(1):6-14, Mar 1993.

[60] L. Kaufman. A Parallel QR algorithm for the Symmetric Tridiagonal Eigenvalue Problem. *Journal of Parallel and Distributed Computing*, 23:429-434, 1994.

[61] S.G. Kratzer. Sparse QR Factorization on a Massively Parallel Computer. *The Journal of Supercomputing*, 6:237-255, 1992.

[62] B. Kumar, P. Sadayappan and C.H. Huang. On Sparse Matrix Reordering for Parallel Factorization. In *Int'l Conference on Supercomputing, ICS'94*, ACM Press, pages 431-438. Manchester, Jul 1994.

[63] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh. Basic Linear Algebra Subprograms for FORTRAN Usage. *ACM Transactions on Mathematical Software*, 5:308-325, 1979.

[64] D.E. Lenoski and W.D. Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann Publishers, 1995.

[65] J.G. Lewis, D.J. Pierce and D.K. Wah. *Multifrontal Householder QR Factorization.* Technical Report ECA-TR-127, Boeing Computer Services, Seattle, 1989.

[66] P. Matstoms. Parallel Sparse QR Factorization on Shared Memory Architectures. *Parallel Computing*, 21:473-486, 1995.

[67] P. Matstoms. Sparse QR Factorization in Matlab. *ACM Transactions on Mathematical Software*, 20(1):136-159, Mar 1994.

[68] A. Matsumoto, D.S. Han and T. Tsuda. Alias Analysis of Pointers in Pascal and Fortran 90: Dependence Analysis between Pointer References. *Acta Informatica*, 33:99-130, 1996.

[69] O.A. McBryan. An Overview of Message Passing Environments. *Parallel Computing*, 20:417-444, 1994.

[70] U. Meier, G. Skinner and J. Gunnels. *A Collection of Codes for Sparse Matrix Computations.* Technical Report 1134, CSRD, University of Illinois at Urbana-Champaign, 1991.

[71] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Version 1.1, Jun 1995.

[72] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Nov 1996.

[73] A. Müller and R. Rühl. Extending High-Performance Fortran for the Support of Unstructured Computations. In *Ninth ACM Int'l Conference on Supercomputing*, pages 127-136. Barcelona, Spain, 1995.

[74] B.K. Natarajan. Sparse Approximate Solutions to Linear Systems. *SIAM J. Computing*, 24(2):227-234, Apr 1995.

[75] T. Ostromsky, P.C. Hansen and Z. Zlatlev. A Parallel Sparse QR-Factorization Algorithm. In *Second Int'l Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering Science, PARA'95*, Springer-Verlag LNCS 1041, pages 462-472. Lyngby, Denmark, Aug 1995.

[76] D.M. Pase, T. McDonald and A. Meltzer. The CRAFT Fortran Programming Model. *Scientific Programming*, 3:227-253, 1994.

[77] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung and D.A. Schouten. The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran. In *Second Workshop on Languages and Compilers for Parallel Computing*, pages 423-453, 1989.

[78] R. Ponnusamy, Y. Hwang, R. Das, J. Saltz, A. Choudhary and G. Fox. Supporting Irregular Distributions Using Data-Parallel Language. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(1):12-24, 1995.

[79] R. Ponnusamy, J. Saltz, A. Choudhary, Y. Hwang and G. Fox. Runtime Support and Compilation Methods for User-Specified Data Distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815-831, 1995.

[80] A. Pothen and P. Raghavan. Distributed Orthogonal Factorization: Givens and Householder Algorithms. *SIAM J. Sci. Stat. Computing*, 10:1113-1134, 1989.

[81] P. Raghavan. Distributed Sparse Gaussian Elimination and Orthogonal Factorization. *SIAM J. Sci. Computing*, 16(6):1462-1477, Nov 1995.

[82] K.A. Remington and R. Pozo. *NIST Sparse BLAS User's Guide.* National Institute of Standards and Technology, Jul 1996.

[83] T.H. Robey and D.L. Sulsky. Row Ordering for a Sparse QR Decomposition. *SIAM J. Matrix Anal. Appl.*, 15(4):1208-1225, Oct 1994.

[84] L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *Parallel Computing*, 21(4):583-605, 1995.

[85] Y. Saad. *SPARSKIT: a Basic Tool Kit for Sparse Matrix Computation.* Technical Report 1029, CSRD, University of Illinois at Urbana-Champaign, 1990.

[86] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y. Hwang, M. Uysal and R. Das. *A Manual for the CHAOS Runtime Library.* Technical Report CS-TR-3437, Dep. Computer Science, University of Maryland, 1995.

[87] T. Schreiber, P. Otto and F. Hofmann. A New Efficient Parallelization Strategy for the QR Algorithm. *Parallel Computing*, 20:63-75, 1994.

[88] S.L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Seventh Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VII*, pages 26-37. Cambridge, Massachusetts, Oct 1996.

[89] D. Sima, T. Fountain and P. Kacsuk. *Advanced Computer Architectures. A Design Space Approach.* Addison-Wesley Publishing Company, 1997.

[90] A.F. van der Stappen, R.H. Bisseling and J.G.G. van de Vorst. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853-879, Jul 1993.

[91] T. Suganuma, H. Komatsu and T. Nakatani. Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines. In *Tenth Int'l Conference on Supercomputing*, pages 18-25. Philadelphia, 1996.

[92] V.S. Sunderam, G.A. Geist, J. Dongarra and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences and Trends. *Parallel Computing*, 20:531-545, 1994.

[93] Thinking Machines Corporation. *CM Fortran Language Reference Manual*, 1994.

[94] J. Touriño and R. Doallo. An Approach to Decrease Fill-in in Sparse Orthogonalizations on an MIMD Computer. In *Sixth Parallel Computing Workshop, PCW'96*, Fujitsu Laboratories Ltd., pages S-B-1 to S-B-5. Kawasaki, Japan, Nov 1996.

[95] J. Touriño and R. Doallo. Local Pivoting for Sparse Factorizations on Multiprocessors. In *Seventh Parallel Computing Workshop, PCW'97*, Ed. Australian National University, pages S-B-1 to S-B-5. Canberra, Australia, Sep 1997.

[96] J. Touriño and R. Doallo. *Parallel Sparse Orthogonalization on an MIMD Distributed Memory Computer.* Technical Report TR-UDC-DES-96/01, Dep. of Electrónica y Sistemas, University of La Coruña, Spain, 1996.

[97] J. Touriño, R. Doallo, R. Asenjo, O. Plata and E.L. Zapata. Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In. In *Sixth Workshop on Compilers for Parallel Computers, CPC'96*, Special Issue of the Volume Konferenzen des Forschungszentrums Jülich, Vol.21, pages 151-168. Aachen, Germany, Dec 1996.

[98] J. Touriño, R. Doallo and E.L. Zapata. Sparse Givens QR Factorization on a Multiprocessor. In *Second Int'l Conference on Massively Parallel Computing Systems, MPCS'96*, IEEE Computer Society Press, pages 524-529. Ischia, Italy, May 1996.

[99] G.P. Trabado and E.L. Zapata. Data-Parallel Language Extensions for Exploiting Locality in Irregular Problems. In *Workshop on Languages and Compilers for Parallel Computing, LCPC'97*. Minnesota, Aug 1997.

[100] G.P. Trabado and E.L. Zapata. Exploiting Locality on Parallel Sparse Matrix Computations. In *Third Euromicro Workshop on Parallel and Distributed Processing, PDP'95*, IEEE Computer Society Press, pages 2-9. San Remo, Italy, 1995.

[101] M. Ujaldón, E.L. Zapata, B. Chapman and H. Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068-1083, Oct 1997.

[102] M. Ujaldón, E.L. Zapata, S.D. Sharma and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *Journal of Parallel and Distributed Computing*, 38:256-266, 1996.

[103] L.C. Waring and M. Clint. Parallel Gram-Schmidt Orthogonalisation on a Network of Transputers. *Parallel Computing*, 17:1043-1050, 1991.

[104] D.S. Watkins. Forward Stability and Transmission of Shifts in the QR Algorithm. *SIAM J. Matrix Anal. Appl.*, 16(2):469-487, Apr 1995.

[105] D.S. Wise and J. Franco. Costs of Quadtree Representation of Nondense Matrices. *Journal of Parallel and Distributed Computing*, 9:282-296, 1990.

[106] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[107] E.L. Zapata, J.A. Lamas, F.F. Rivera and O. Plata. Modified Gram-Schmidt QR Factorization on Hypercube SIMD Computers. *Journal of Parallel and Distributed Computing*, 12:60-69, 1991.

[108] B.B. Zhou and R.P. Brent. *Parallel Implementation of QRD Algorithms on the Fujitsu AP1000.* Technical Report TR-CS-93-12, Computer Sciences Laboratory, The Australian National University, Canberra, Australia, Nov 1993.

[109] H. Zima, H.J. Bast and M. Gerndt. SUPERB - a Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1-18, 1988.

[110] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald. *Vienna Fortran - A Language Specification.* Technical Report ACPC-TR92-4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.

[111] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991

[112] Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, 1991.