

---

# PARALLEL NUMERICAL COMPUTATION WITH APPLICATIONS

# PARALLEL NUMERICAL COMPUTATION WITH APPLICATIONS

Edited by  
Tianruo Yang

**Kluwer Academic Publishers**  
Boston/Dordrecht/London

# Contents

Preface	ix
Part I Parallel Numerical Algorithms	
1	
PSPASES: Scalable Parallel Direct Solver for Sparse Systems	3
<i>Mahesh V. Joshi, George Karypis, Vipin Kumar, Anshul Gupta and Fred Gustavson</i>	
1.1 Introduction	4
1.2 Algorithms	5
1.3 Library Implementation and Experimental Results	11
References	13
2	
Parallel Factorization Algorithms	19
<i>Jaeyoung Choi</i>	
2.1 Introduction	19
2.2 PoLAPACK LU Factorization Algorithm	21
2.3 Implementation of PoLAPACK LU Factorization	25
2.4 PoLAPACK QR Factorization	28
2.5 Conclusions	29
References	31
3	
Parallel Solution of Stiff ODE	33
<i>Thomas Rauber, Gudula Rünger</i>	
3.1 Introduction	34
3.2 Implicit Runge–Kutta Methods	35
3.3 Parallel Implementations	39
3.4 Runtime Tests	42
3.5 Conclusions	44
3.6 Acknowledgements	47
References	47
	v

4		
Parallel Space Decomposition		53
<i>Andreas Frommer, Rosemary A. Renaut</i>		
4.1	Introduction	54
4.2	Parallel Space decomposition	55
4.3	Convergence Theory	57
4.4	The Constrained PVD Algorithm	60
References		61
5		
Parallel Biconjugate Gradient Method		63
<i>H. Martin Bücker, Manfred Sauren</i>		
5.1	Introduction	64
5.2	Unsymmetric Lanczos algorithm	65
5.3	A parallel variant of BCG	67
5.4	A rescaled version of the original BCG	71
5.5	Numerical experiments	72
5.6	Concluding Remarks	74
5.7	Acknowledgments	75
References		75
6		
Jacobi ordering for Multiple-port Hypercubes		77
<i>Dolors Royo, Miguel Valero-García and Antonio González</i>		
6.1	Introduction	77
6.2	Preliminaries	78
6.3	The <i>Degree-4</i> ordering	83
6.4	A systematic transformation of hamiltonian paths	84
6.5	<i>Permuted-D4</i> ordering	85
6.6	Performance evaluation	87
6.7	Conclusions	87
References		87
7		
Performance Analysis of IQMR Method		89
<i>Tianruo Yang, Hai-Xiang Lin</i>		
7.1	Introduction	90
7.2	Lanczos Process Based on Three-Term Recurrence	91
7.3	Lanczos Process Based on Coupled Two-Term Recurrence	92
7.4	The Improved Quasi-Minimal Residual Method	93
7.5	The Performance Model	95
7.6	Theoretical Performance Analysis	96
7.7	Conclusion	100
References		100

## Part II    Parallel Numerical Applications

8		
Parallel algorithm for 3D elasticity problems		105
<i>Krassimir Georgiev</i>		
8.1 Introduction		105
8.2 The Problem		106
8.3 How the Domain Decomposition preconditioner is constructed		108
8.4 The parallel solver		109
8.5 Numerical experiments		112
8.6 Conclusions and outlook		115
References		116
9		
Distributed Control Parallelism		119
<i>Denitza T. Krasteva, Layne T. Watson, Chuck A. Baker, Bernard Grossman, William H. Mason, Raphael T. Haftka</i>		
9.1 HSCT Configuration Optimization		122
9.2 Problem Definition		125
9.3 Description of Algorithms		128
9.4 Parallel Implementation		132
9.5 Discussion of Results		133
9.6 Summary		137
References		137
10		
Semiconductor Device and Process Simulation		141
<i>Olaf Schenk<sup>†</sup>, Klaus Gärtner<sup>‡</sup>, Bernhard Schmithüsen<sup>†</sup> and Wolfgang Fichtner<sup>†</sup></i>		
10.1 Introduction: Semiconductor Device Simulation and Algorithms		142
10.2 High computational performance for Parallel Sparse LU Factorization		143
10.3 Comparing Algorithms and Machines for complete Semiconductor Device Simulations		147
10.4 Conclusion		152
References		154
11		
Parallel Overlapping Meshes		159
<i>Jacek Rokicki Dimitris Drikakis Jerzy Majewski Jerzy Zóttak</i>		
11.1 Introduction		160
11.2 Implicit unfactored solution		161
11.3 The overlapping-mesh algorithm		161
11.4 Numerical results		165
11.5 Parallel implementation		169
11.6 Conclusions		173
References		174
12		
The Optimized Order 2 Method		177
<i>Caroline Japhet Frédéric Nataf Francois Rogier</i>		
12.1 The Optimized Order 2 Method		178
12.2 Numerical results		181

12.3	Remark	183
12.4	Conclusion	183
References		189
13		
A Parallel Algorithm for a Lubrication Problem		191
<i>M. Arenaz, R. Doallo and J. Touriño C. Vázquez</i>		
13.1	Description of the Problem	192
13.2	The Numerical Algorithm	194
13.3	The Parallelization Process	195
13.4	Experimental Results	197
13.5	Conclusions and Future Work	198
References		201
14		
Parallel Computing of Cavity Flow		203
<i>Hui Wan, Shanwu Wang, Yanxiang Wang, Xiyun Lu and Lixian Zhuang</i>		
14.1	Introduction	203
14.2	Mathematical Formulation and Numerical Methods	204
14.3	Parallel Implementation	207
14.4	Results and Discussion on 3D Cavity Flow	208
14.5	Concluding Remarks on MPI	212
References		213
15		
Electromagnetic Scattering with Boundary Integral Method		215
<i>Thierry Jacques, Laurent Nicolas and Christian Vollaire</i>		
15.1	Introduction	216
15.2	Impact of the Parallel Computing in The Electrical Engineering	216
15.3	Integral Equation Method in the Parallel Computation	218
15.4	Boundary Integral Formulation for Electromagnetic Scattering Problems	221
15.5	Parallel Computation	223
15.6	Numerical Results	225
15.7	Conclusion	228
References		228

## Preface

Scientific and engineering computing has become a key technology which will play an important part in determining, or at least shaping, future research and development activities in many academic and industrial branches. The increasing processing speed and expanded storage capacity of modern high-performance computers, together with new advanced numerical methods and programming techniques, have significantly improved the ability to solve complex scientific and engineering problems. In order to efficiently tackle those complex scientific and engineering problems we are facing today, it is very necessary to recourse to parallel computing.

The application of parallel and distributed computers to numerically intensive problems brings together several issues that do not appear in conventional computing. New algorithms, codes and parallel computing techniques are required in order to effectively exploit the use of these novel computer architectures and achieve very modest improvement of performance on vector and parallel processors.

The workshop, Frontiers of Parallel Numerical Computations and Applications, is organized in the IEEE 7th Symposium on the Frontiers on Massively Parallel Computers (Frontiers'99) at Annapolis, Maryland, February 20-25, 1999. The main purpose of this workshop is to bring together computer scientists, applied mathematicians and researchers to present, discuss and exchange novel ideas, new results, work in progress and advancing state-of-the-art techniques in the area of parallel and distributed computing for numerical and computational optimization problems in scientific and engineering applications.

This book contains all papers presented at the above workshop with some invited papers from the leading researchers around the world. The papers included in this book cover a broad spectrum of topics on parallel numerical computation with applications such as development of advanced parallel numerical and computational optimization methods, novel parallel computing techniques and performance analysis for above methods, applications to numerical fluid mechanics, material sciences, applications to signal and image processing, dynamic systems, semiconductor technology, and electronic circuits and systems design etc.

The editor is grateful to all authors for their excellent contributions and Alex Greene of Kluwer Academic Publishers for his patience and support to make this book possible.

We hope this book will increase the awareness of the potential of new parallel numerical computation and applications among scientists and engineers.

TIANRUO YANG, LINKÖPING, SWEDEN, 1999



# **I   PARALLEL NUMERICAL COMPUATION**



# 1 PSPASES: BUILDING A HIGH PERFORMANCE SCALABLE PARALLEL DIRECT SOLVER FOR SPARSE LINEAR SYSTEMS\*

Mahesh V. Joshi, George Karypis, Vipin Kumar,

*Department of Computer Science,  
University of Minnesota, Minneapolis, USA*  
{mjoshi,karypis,kumar}@cs.umn.edu

Anshul Gupta and Fred Gustavson

*Mathematical Sciences Department,  
IBM T. J. Watson Research Center,  
Yorktown Heights, New York, USA*  
{anshul,gustav}@watson.ibm.com

**Abstract:** Many problems in engineering and scientific domains require solving large sparse systems of linear equations, as a computationally intensive step towards the final solution. It has long been a challenge to develop efficient parallel formulations

---

\*This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAG55-98-1-0441, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPARC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: <http://www.cs.umn.edu/~kumar>.

of sparse direct solvers due to several different complex steps involved in the process. In this paper, we describe PSPASES, one of the first efficient, portable, and robust scalable parallel solvers for sparse symmetric positive definite linear systems that we have developed. We discuss the algorithmic and implementation issues involved in its development; and present performance and scalability results on Cray T3E and SGI Origin 2000. PSPASES could solve the largest sparse system (1 million equations) ever solved by a direct method, with the highest performance (51 GFLOPS for Cholesky factorization) ever reported.

## 1.1 INTRODUCTION

Solving large sparse systems of linear equations is at the heart of many engineering and scientific computing applications. There are two methods to solve these systems - direct and iterative. Direct methods are preferred for many applications because of various properties of the method and the nature of the application. A wide class of sparse linear systems arising in practice have a symmetric positive definite (SPD) coefficient matrix. The problem is to compute the solution to the system  $Ax = b$ , where  $A$  is a sparse and SPD matrix. Such a system is commonly solved using Cholesky factorization. A direct method of solution consists of four consecutive phases viz. ordering, symbolic factorization, numerical factorization and solution of triangular systems. During the ordering phase, a permutation matrix  $P$  is computed so that the matrix  $PAP^T$  will incur a minimal fill during the factorization phase. During the symbolic factorization phase, the non-zero structure of the triangular Cholesky factor  $L$  is determined. The symbolic factorization phase exists in order to increase the performance of the numerical factorization phase. The necessary operations to compute the values in  $L$  that satisfy  $PAP^T = LL^T$ , are performed during the phase of numerical factorization. Finally, the solution to  $Ax = b$  is computed by solving two triangular systems viz.  $Ly = b'$  followed by  $L^T x' = y$ , where  $b' = Pb$  and  $x' = Px$ . Solving the former system is called forward elimination and the latter process of solution is called backward substitution. The final solution,  $x$ , is obtained using  $x = P^T x'$ .

In this paper, we describe one of the first scalable and high performance parallel direct solvers for sparse linear systems involving SPD matrices. At the heart of this solver is a highly parallel algorithm based on multifrontal Cholesky factorization we recently developed [2]. This algorithm is able to achieve high computational rates (of over 50 GFLOPS on a 256 processor Cray T3E) and it successfully parallelizes the most computationally expensive phase of the sparse solver. Fill reducing ordering is obtained using a parallel formulation of the multilevel nested dissection algorithm [5] that has been found to be effective in producing orderings that are suited for parallel factorization. For symbolic factorization and solution of triangular systems, we have developed parallel algorithms that utilize the same data distribution as used by the numerical factorization algorithm. Both algorithms are able to effectively parallelize the corresponding phases. In particular, our parallel forward elimination and parallel backward substitution algorithms are scalable and achieve high computational rates [3].

We have integrated all these algorithms into a software library, PSPASES (Parallel SPArse Symmetric dirEct Solver). It uses the MPI library for communication, making it portable to a wide range of parallel computers. Furthermore, high computational rates are achieved using serial BLAS routines to perform the computations at each processor. Various easy-to-use functional interfaces, and internal memory management are among other features of PSPASES.

In the following, first we briefly describe the parallel algorithms for each of the phases, and their theoretical scalability behavior. Then, we describe the functional interfaces and features of the PSPASES library. Finally, we demonstrate its performance and scalability for a wide range of matrices on Cray T3E and SGI Origin 2000 machines.

## 1.2 ALGORITHMS

PSPASES implements scalable parallel algorithms in each of the phases. In essence, the algorithms are driven by the elimination tree structure of  $A$  [8]. The ordering phase computes an ordering with the aims of reducing the fill-ins and generating a balanced elimination tree. Symbolic factorization phase traverses the elimination tree in a bottom-up fashion to compute the non-zero structure of  $L$ . Numerical factorization phase is based on a highly parallel multifrontal algorithm. This phase determines the internal data distribution of  $A$  and  $L$  matrices. The triangular solve phase also uses multifrontal algorithms. Following subsections give more details of each of the phases.

### 1.2.1 Parallel Ordering

We use our multilevel nested dissection based algorithm for computing fill-reducing ordering.

The matrix  $A$  is converted to its equivalent graph, such that each row  $i$  is mapped to a vertex  $i$  of the graph and each nonzero  $a_{ij}$  ( $i \neq j$ ) is mapped to an edge between vertices  $i$  and  $j$ . A parallel graph partitioning algorithm [5] is used to compute a high quality  $p$ -way partition, where  $p$  is the number of processors. The graph is then redistributed according to this partition. This pre-partitioning phase helps to achieve high performance for computing the ordering, which is done by computing the  $\log p$  levels of the elimination tree concurrently. At each level, multilevel paradigm is used to partition the vertices of the graph. The set of nodes lying at the partition boundary forms a *separator*. Separators at all levels are stored. When the graph is separated into  $p$  parts, it is redistributed among the processors such that each processor receives a single subgraph, which it orders using the multiple minimum degree (MMD) algorithm. Finally, a global numbering is imposed on all nodes such that nodes in each separator and each subgraph are numbered consecutively, respecting the MMD ordering.

Details of the multilevel paradigm can be found in [5]. Briefly, it involves gradual coarsening of a graph to a few hundred vertices, then partitioning this smaller graph, and finally, projecting the partitions back to the original graph by gradually refining them. Since the finer graph has more degrees of freedom, such refinements improve the quality of the partition.

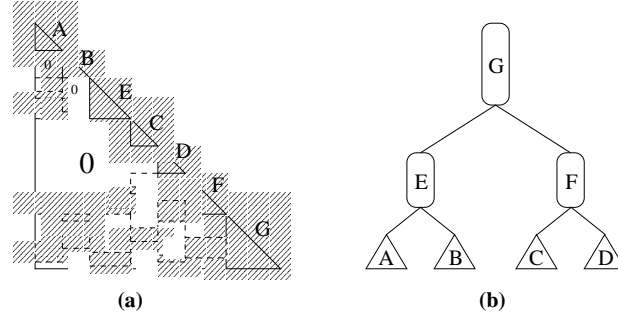


Figure 1.1 Ordering a system for 4 processors. (a) Structure of a reordered matrix, (b) Separator tree corresponding to it.

When a matrix is reordered according to the ordering generated by above algorithm, it has a format similar to that shown in Figure 1.1(a). The separator tree corresponding to the reordered graph has a structure of the form shown in Figure 1.1(b).

### 1.2.2 Parallel Numerical Factorization

We will first describe the parallel numerical factorization phase, because algorithm used here decides the data distribution of  $L$ , and hence drives the symbolic factorization algorithm. We use our highly scalable algorithm [2] that is based on the multifrontal algorithm [9].

Given a sparse matrix and the associated elimination tree, the multifrontal algorithm can be recursively formulated as follows. Consider an  $N \times N$  matrix  $A$ . The algorithm performs a postorder traversal of the elimination tree associated with  $A$ . There is a frontal matrix  $F^k$  and an update matrix  $U^k$  associated with any node  $k$ . The row and column indices of  $F^k$  correspond to the indices of row and column  $k$  of  $L$ , the lower triangular Cholesky factor, in increasing order. In the beginning,  $F^k$  is initialized to an  $(s + 1) \times (s + 1)$  matrix, where  $s + 1$  is the number of non-zeros in the lower triangular part of column  $k$  of  $A$ . The first row and column of this initial  $F^k$  is simply the upper triangular part of row  $k$  and the lower triangular part of column  $k$  of  $A$ . The remainder of  $F^k$  is initialized to all zeros.

After the algorithm has traversed all the subtrees rooted at a node  $k$ , it ends up with a  $(t + 1) \times (t + 1)$  frontal matrix  $F^k$ , where  $t$  is the number of non-zeros in the strictly lower triangular part of column  $k$  in  $L$ . The row and column indices of the final assembled  $F^k$  correspond to  $t + 1$  (possibly) noncontiguous indices of row and column  $k$  of  $L$  in increasing order. If  $k$  is a leaf in the elimination tree of  $A$ , then the final  $F^k$  is the same as the initial  $F^k$ . Otherwise, the final  $F^k$  for eliminating node  $k$  is obtained by merging the initial  $F^k$  with the update matrices obtained from all the subtrees rooted at  $k$  via an extend-add operation. The extend-add is an associative and commutative operator on two update matrices such the index set of the result is the union of the index sets of the original update matrices. Each entry in the original update matrices is mapped onto some location in the accumulated matrix. If entries from both matrices overlap on a location, they are added. Empty entries are assigned a value of zero. After  $F^k$  has been assembled, a single step of the standard dense Cholesky

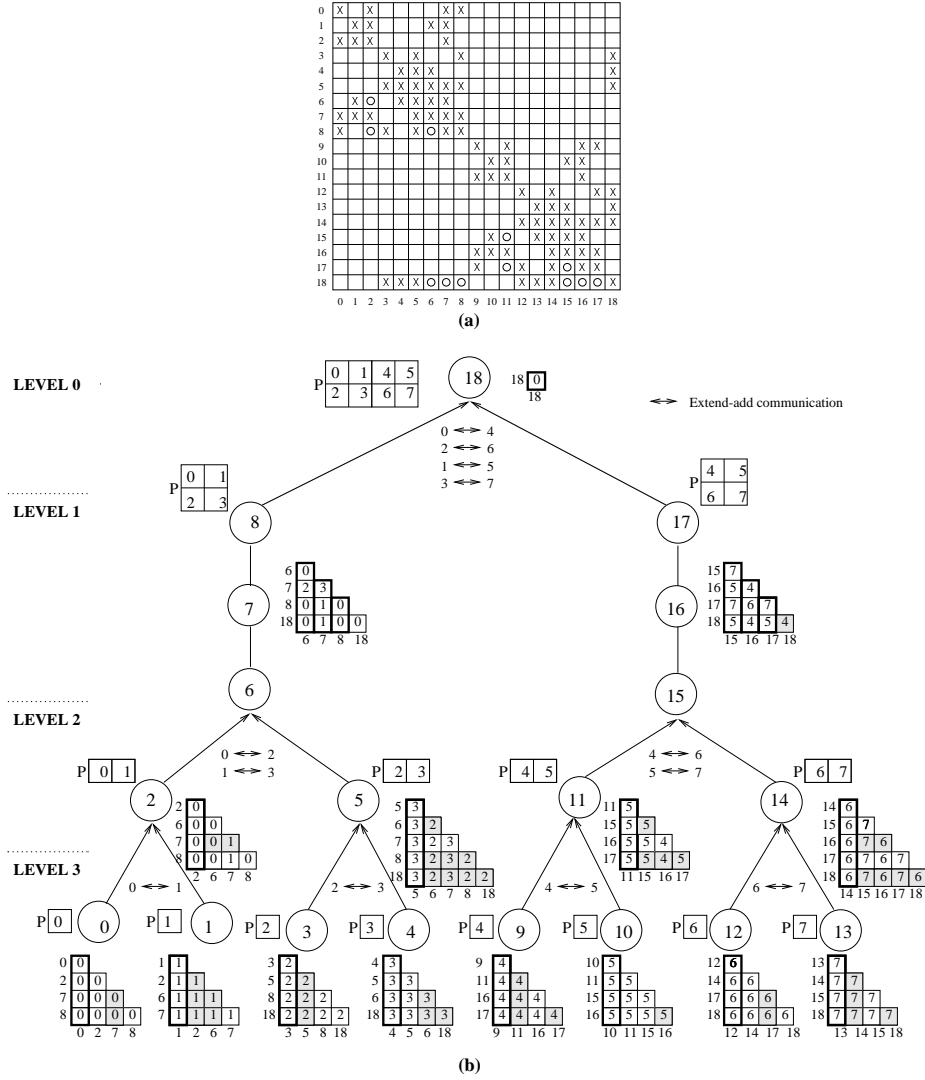


Figure 1.2 (a). An example symmetric sparse matrix. The non-zeros of  $A$  are shown with symbol “x” in the upper triangular part and non-zeros of  $L$  are shown in the lower triangular part with fill-ins denoted by the symbol “o”. (b). The process of parallel multifrontal factorization using 8 processors. At each supernode, the factored frontal matrix, consisting of columns of  $L$  (thick columns) and update matrix (remaining columns), is shown.

factorization is performed with node  $k$  as the pivot. At the end of the elimination step, the column with index  $k$  is removed from  $F^k$  and forms the column  $k$  of  $L$ . The remaining  $t \times t$  matrix is called the update matrix  $U^k$  and is passed on to the parent of  $k$  in the elimination tree.

A collection of consecutive nodes in the elimination tree, each with only one child, is called a *supernode*. Nodes in each supernode are collapsed together to form a *supernodal elimination tree*, which corresponds to the separator tree [Figure 1.1(b)] in the top  $\log p$  levels.

The serial multifrontal algorithm can be extended to operate on this supernodal tree by extending the single node operations performed while forming and factoring the frontal matrix. The frontal matrix corresponding to a supernode with  $l$  nodes is obtained by merging the frontal matrices of the individual nodes, and the first  $l$  columns of this frontal matrix are factored during the factorization of this supernode.

In our parallel formulation of the multifrontal algorithm, we assume that the supernodal tree is binary in the top  $\log p$  levels<sup>1</sup>. The portions of this binary supernodal tree are assigned to processors using a subtree-to-subcube strategy illustrated in Figure 1.2(b), where eight processors are used to factor the example matrix of Figure 1.2(a). The subcubes of processors working on various subtrees are shown in the form of a logical mesh labeled with P. The frontal matrix of each supernode is distributed among this logical mesh using a bitmask based block-cyclic scheme [2]. Figure 1.2(b) shows such a distribution for unit blocksize. This distribution ensures that the extend-add operations required by the multifrontal algorithm can be performed in parallel with each processor exchanging roughly half of its data *only* with its partner from the other subcube. Figure 1.2(b) shows the parallel extend-add process by showing the pairs of processors that communicate with each other. Each processor sends out the shaded portions of the update matrix to its partner. The parallel factor operation at each supernode is a pipelined implementation of the dense column Cholesky factorization algorithm.

### 1.2.3 Parallel Symbolic Factorization

During the symbolic factorization phase the non-zero structure of the factor matrix  $L$  is determined. The serial algorithm to generate the structure of  $L$  performs a postorder traversal of the elimination tree. At each node  $k$ , the  $L$  indices of all its children node (excluding the children nodes themselves) and the  $A$  indices of  $k$  are merged together to form the  $L$  indices of node  $k$ .

This algorithm can be effectively parallelized using the same subtree-to-subcube mapping used by the numerical factorization algorithm. The basic structure of the algorithm is illustrated in Figure 1.3. Initially, matrix  $A$  is distributed such that the columns of  $A$  of each supernode are distributed using the same bitmask based block-cyclic distribution required by the numerical factorization phase<sup>2</sup>. Now, the non-zero structure of  $L$  is determined in a bottom-up fashion. First the non-zero structure of the leaf nodes is determined and it is sent upwards in the tree, to the processors that store the next level supernode. These processors determine the non-zero structure of their supernode and merge it with the non-zero structure received from their children nodes. For example, consider the computation involved in determining the structure

<sup>1</sup>The separator tree obtained from the recursive nested dissection parallel ordering algorithm used in our solver yields such a binary tree (Fig. 1.1(b)).

<sup>2</sup>This distribution is performed after computing ordering.



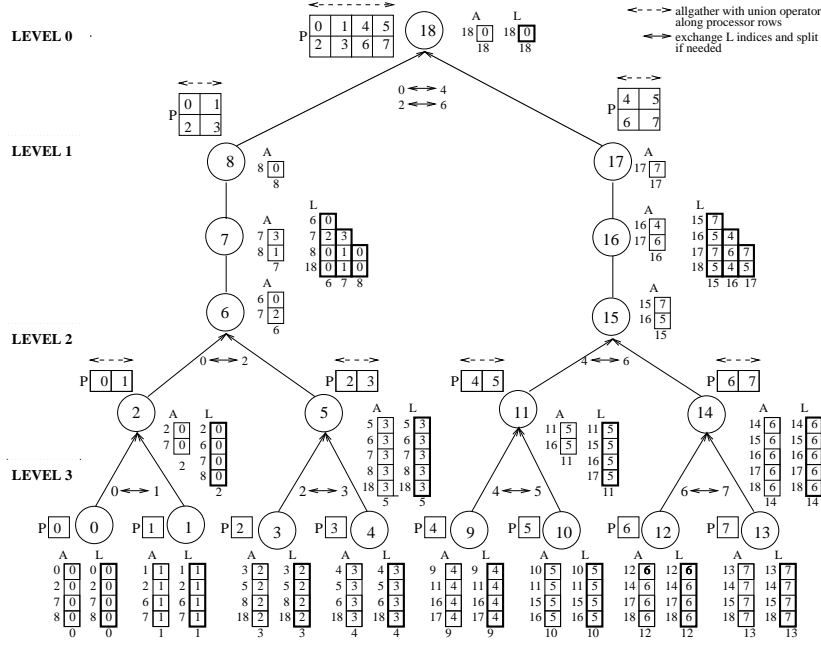


Figure 1.3 Parallel Symbolic Factorization

of  $L$  for the supernode consisting of the nodes  $\{6,7,8\}$  which are distributed among a  $2 \times 2$  processor grid. First, the processors determine the non-zero structure of  $L$  by performing a union along the rows of the grid to collect the distinct row indices of the columns of  $A$  corresponding to the nodes  $\{6,7,8\}$ . The result is  $\{6,8\}$  and  $\{7\}$  at the first and second row of processors, respectively. Now the non-zero structures of the children nodes are received (excluding the nodes themselves). Since these non-zero structures are stored in the two  $1 \times 2$  sub-grids of the  $2 \times 2$  grid, this information is sent using a similar communication pattern described in Section 1.2.2, however only the processors in the first column of the subgrids need to communicate. In particular, processor 0 splits the list  $\{6,7,8\}$  into two parts  $\{6,8\}$  and  $\{7\}$ , retains  $\{6,8\}$  and sends  $\{7\}$  to processor 2. Similarly processor 2 splits the list  $\{6,7,8,18\}$  into two parts  $\{6,8,18\}$  and  $\{7\}$ , retains  $\{7\}$  and sends  $\{6,8,18\}$  to processor 0. Now processors 0 and 2 merge these lists. In particular, processor 0 merges  $\{6,8\}$ ,  $\{6,8\}$  and  $\{6,8,18\}$  and processor 2 merges  $\{7\}$ ,  $\{7\}$  and  $\{7\}$ .

### 1.2.4 Parallel Triangular Solve

During the phase of solving triangular systems, a forward elimination  $Ly = b'$ , where  $b' = Pb$ , is performed followed by a backward substitution  $L^T x' = y$  to determine the solution  $x = P^T x'$ . Our parallel algorithms for this phase are guided by the supernodal elimination tree. They use the same subtree-to-subcube mapping and the same two-dimensional distribution of the factor matrix  $L$  as used in the numerical factorization.

Figure 1.4(a) illustrates the parallel formulation of the forward elimination process. The right hand side vector  $b'$ , is distributed to the processors that own the corresponding diagonal blocks of the  $L$  matrix as shown in the shaded blocks in Figure 1.4(a). The computation proceeds in a bottom-up fashion. Initially, for each leaf node  $k$ , the solution  $y_k$  is computed and is used to form the update vector  $\{l_{ik}y_k\}$  (denoted by "U" in Figure 1.4(a)). The elements of this update vector need to be subtracted from the corresponding elements of  $b'$ , in particular  $l_{ik}y_k$  will need to be subtracted from  $b'_i$ . However, our algorithm uses the structure of the supernodal tree to accumulate these updates upwards in the tree and subtract them only when the appropriate node is being processed. For example consider the computation involved while processing the supernode  $\{6,7,8\}$ . First the algorithm merges the update vectors from the children supernodes to obtain the combined update vector for indices  $\{6,7,8,18\}$ . Note that the updates to the same  $b'$  entries are added up. Then it performs forward elimination to compute  $y_6, y_7$  and  $y_8$ . This computation is done using a two dimensional pipelined dense forward elimination algorithm. At the end of the computation, the update vector on processor 0 contains the updates for  $b'_{18}$  due to  $y_6, y_7$  and  $y_8$  as well as the updates received from supernode  $\{5\}$ . In general, at the end of the computation at each supernode, the accumulated update vector resides on the column of processors that store the last column of the  $L$  matrix of that supernode. This update vector needs to be sent to the processors that store the first column of the  $L$  matrix of the parent supernode. Because of the bitmask based block-cyclic distribution, this can be done by using at most two communication steps [3].

The details of the two-dimensional pipelined dense forward elimination algorithm are illustrated in Figure 1.4(b) for a hypothetical supernode. The solutions are computed by the processors owning diagonal elements of  $L$  matrix and flow down along a column. The accumulated updates flow along the row starting from the first column and ending at the last column of the supernode. The processing is pipelined in the shaded regions and in other regions the updates are accumulated using a reduction operation along the direction of the flow of update vector.

The algorithm for parallel backward substitution is similar, except for two differences. First, the computation proceeds from the top supernode of the tree down to the leaf. Second, the computed solution that gets communicated across the levels of the supernodal tree instead of accumulated updates and this is achieved with at most one communication per processor. Refer to [3] for details.

### 1.2.5 Parallel Runtime and Scalability Analysis

Analyzing a general sparse solver is a difficult problem. We present the analysis for two wide classes of sparse systems arising out of two-dimensional and three-dimensional constant node-degree graphs. We refer to these problems as 2-D and 3-D problems, respectively.

Let a problem of dimension  $N$  be solved using  $p$  processors. Define  $l$  as a level in the supernodal tree with  $l = 0$  for the topmost level. Thus, the number of processors assigned to a level  $l$  supernode is  $p/2^l$ . With a nested dissection based ordering scheme, it has been shown that the average size of a frontal matrix at level  $l$  is  $\theta(N/2^{2l})$  for 2-D problems, and  $\theta(N^{4/3}/2^{4l})$  for 3-D problems. Also, the number of nodes in a level- $l$

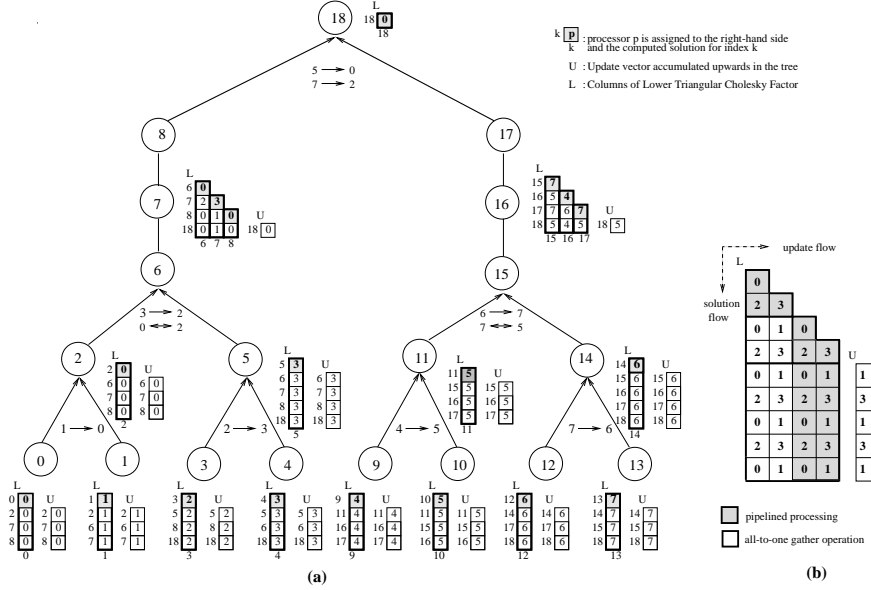


Figure 1.4 Parallel Triangular Solve. (a). Entire process of parallel forward elimination for the example matrix. (b). Processing within a hypothetical supernodal matrix for forward elimination.

supernode is on an average  $\theta(\sqrt{N/2^l})$  and  $\theta((N/2^l)^{2/3})$  for 2-D and 3-D problems, respectively. The number of non-zeros in  $L$  is  $N \log N$  for 2-D problems, and  $N^{4/3}$  for 3-D problems. Using this information, the parallel algorithms presented earlier in this section have been analyzed for their parallel runtime. For details, refer to [1, 3].

Table 1.1 shows the serial runtime complexity, parallel overhead, and isoefficiency functions [7] for all four phases. It can be seen that all the phases are scalable to varying degrees. Since the overall time complexity is dominated by the numerical factorization phase, the isoefficiency function of the entire parallel solver is determined by the numerical factorization phase, and it is  $O(p^{1.5})$  for both 2-D and 3-D problems. As discussed in [7], the isoefficiency function of the dense Cholesky factorization algorithm is also  $O(p^{1.5})$ . Thus our sparse direct solver is as scalable as the dense factorization algorithm.

### 1.3 LIBRARY IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented all the parallel algorithms presented in previous section, and integrated them into a library called PSPASES [4]. This library uses MPI call interface for communication, and BLAS interface for computation within a processor. This makes PSPASES portable to a wide range of parallel computers. Furthermore, using BLAS, it is able to achieve high computational performance especially on the platforms with vendor-tuned BLAS libraries.

*Table 1.1* Complexity Analysis of various phases of our parallel solver for  $N$ -vertex constant node-degree graphs. Notations:  $T_s$  = Serial Runtime Complexity,  $T_o$  = Parallel Overhead =  $pT_p - T_s$ , where  $T_p$  is the Parallel Runtime. IsoEff is the isoefficiency function indicating the rate at which the problem size should increase with the number of processors to achieve same efficiency.

Phase	2-D constant node-degree graph			3-D constant node-degree graph		
	$T_s$	$T_o$	IsoEff	$T_s$	$T_o$	IsoEff
Order	$O(N \log N)$	$O(p^2 \log p)$	$O(p^2 \log p)$	$O(N \log N)$	$O(p^2 \log p)$	$O(p^2 \log p)$
SymFact	$O(N \log N)$	$O(\sqrt{Np} \log p)$	$O(p \log p)$	$O(N^{4/3})$	$O(N^{2/3} \sqrt{p})$	$O(p)$
NumFact	$O(N^{3/2})$	$O(N \sqrt{p})$	$O(p \sqrt{p})$	$O(N^2)$	$O(N^{4/3} \sqrt{p})$	$O(p \sqrt{p})$
TriSolve	$O(N \log N)$	$O(\sqrt{Np})$	$O(p^2 / \log p)$	$O(N^{4/3})$	$O(N^{2/3} p)$	$O(p^2)$

We have incorporated some more features into PSPASES to make it easy-to-use. The library provides simple functional interfaces, and allows user to call its functions from both C and Fortran 90 programs. It uses the concept of a communicator which relieves the user from managing the memory required by various internal data structures. The communicator also allows to solve multiple systems with same sparsity structure, or same system for multiple sets of right hand side matrix,  $B$ . PSPASES also provides the user with useful statistical information such as the load imbalance of the supernodal elimination tree, number of fill-ins (both of which can be used to judge the quality of ordering), operation count for factor operation, and estimation of memory for internal data structures.

We have tested PSPASES on a wide range of parallel machines, for a wide range of sparse matrices. Tables 1.3, 1.4, 1.5 and 1.6 give experimental results obtained on a 32-processor SGI Origin 2000 and a 256-processor Cray T3E-1200, respectively. We used native versions of both the MPI and BLAS libraries. The notations used in presenting the results are described in Table 1.2. We give numbers for the numerical factorization and triangular solve phases only. The parallel ordering algorithm we use is incorporated into a separate library called ParMetis [6], and PSPASES library functions just call the ParMetis library functions. ParMetis performance numbers are reported elsewhere [6]. The symbolic factorization took relatively very small time, and hence, we do not report those numbers here.

It should be noted that the blocksize of distributing supernodal matrices and the load balance of the supernodal tree played important roles in determining the performance of the solver. Also, since most of today's parallel computers have better communication performance for larger data being communicated, blocksize optimal for numerical factorization need not be optimal for triangular solve phase, which has relatively small data communication per computational block. For the results presented, blocks of 1024 or 4096 double precision numbers worked best for numerical factorization phase.

Table 1.2 Notations used in the presented results

N	:	Dimension of A.
NNZ_LowerA	:	Number of non-zeros in Lower triangular part of A (including diagonal).
np	:	Number of Processors.
NNZ_L	:	Number of non-zeros in L.
Imbal	:	Computational Load Imbalance due to supernodal tree structure.
FAC_OPC	:	Operation count for Numerical Factor phase.
Ntime	:	Numerical Factor Time (in seconds).
Nperf	:	Numerical Factor Performance (in MFLOPS).
Time	:	Redistribution of b and x + Triangular Solve Time for nrhs = 1.

It can be seen from Tables 1.3, 1.4, 1.5 and 1.6 that numerical factorization phase demonstrates high performance and scalability for all the matrices. Triangular solve phase also yields a very high performance, but for larger number of processors it tends to show an unscalable behavior. We profiled various parts of the code to see where the unscalability was appearing and found that it was in the part where  $b$  and  $x'$  are being permuted in parallel ( $b' = Pb$  and  $x = P^T x'$ ). Analyzing it further showed that bad performance of all-to-all operations of underlying MPI library for large chunks of data being exchanged among large number of processors caused such a behavior. The basic algorithms for both forward elimination and backward substitution performed scalably.

The highest performance of 51 GFLOPS was obtained for numerical factorization for the 1-million equation system CUBE100, on 256 processors of Cray T3E-1200. To our knowledge, this is the largest sparse system ever solved by a direct solution method, with the highest performance ever reported.

We have made PSPASES library and more experimental results available via WWW at URL: <http://www.cs.umn.edu/~mjoshi/pspases>.

## References

- [1] A. Gupta. *Analysis and Design of Scalable Parallel Algorithms for Scientific Computing*. PhD thesis, University of Minnesota, Minneapolis, MN, 1995.
- [2] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.
- [3] M. Joshi, A. Gupta, G. Karypis, and V. Kumar. A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems. In *Fourth Int. Conference on High Performance Computing (HiPC'97)*, Bangalore, India, 1997.

*Table 1.3* Performance for factorization and solve phases on SGI Origin 2000 (1)

np	NNZ_L	Imbal	FAC_OPC	Ntime	Nperf	Ttime
matrix : MDUAL2, N: 988605, NNZ_LowerA: 2.9357E+06						
4	1.77e+8	1.29	3.20e+11	685.420	466.56	8.899
8	1.90e+8	1.66	3.98e+11	490.000	811.57	5.151
16	1.96e+8	1.32	4.09e+11	256.588	1592.17	3.217
32	1.92e+8	1.43	3.63e+11	140.110	2590.74	2.399
matrix : AORTA, N: 577771, NNZ_LowerA: 1.7067E+06						
4	7.64e+7	1.48	7.95e+10	163.648	485.87	4.475
8	7.79e+7	1.74	8.47e+10	92.727	913.89	2.566
16	8.20e+7	2.59	9.46e+10	79.850	1185.14	1.633
32	8.49e+7	2.46	1.02e+11	52.877	1936.23	1.314
matrix : CUBE65, N: 274625, NNZ_LowerA: 1.0858E+06						
4	1.33e+8	1.11	4.04e+11	737.626	547.68	4.878
8	1.32e+8	1.15	4.04e+11	485.296	832.73	2.550
16	1.36e+8	1.27	4.19e+11	306.904	1363.67	1.423
32	1.40e+8	1.27	4.27e+11	137.730	3100.99	1.379

Table 1.4 Performance for factorization and solve phases on SGI Origin 2000 (2)

np	NNZ_L	Imbal	FAC_OPC	Ntime	Nperf	Ttime
matrix : S3DKQ4M2, N: 90449, NNZ_LowerA: 2.4600E+06						
2	1.90e+7	1.05	8.87e+09	22.418	395.49	0.946
4	1.92e+7	1.10	8.72e+09	12.351	705.93	0.570
8	2.17e+7	1.44	1.29e+10	12.249	1051.67	0.583
16	2.18e+7	1.43	1.25e+10	7.357	1702.44	0.613
32	2.29e+7	1.58	1.36e+10	6.404	2124.37	0.596
matrix : COPTER2, N: 55476, NNZ_LowerA: 4.0771E+05						
2	9.88e+6	1.01	5.84e+09	14.433	404.61	0.624
4	1.02e+7	1.08	6.47e+09	9.102	710.73	0.438
8	1.06e+7	1.50	6.85e+09	6.960	984.09	0.397
16	1.11e+7	1.39	7.38e+09	4.616	1599.81	0.380
32	1.19e+7	1.39	8.29e+09	4.121	2011.69	0.422
matrix : BCSSTK18, N: 11948, NNZ_LowerA: 8.0500E+04						
2	6.18e+5	1.35	1.01e+08	0.672	149.58	0.084
4	6.54e+5	1.88	1.17e+08	0.496	234.91	0.051
8	6.71e+5	1.47	1.21e+08	0.272	444.55	0.034
16	8.10e+5	2.30	1.82e+08	0.302	602.09	0.034
32	7.56e+5	1.95	1.45e+08	0.270	537.83	0.051

*Table 1.5* Performance for factorization and solve phases on Cray T3E-1200 (1)

np	NNZ_L	Imbal	FAC_OPC	Ntime	Nperf	Ttime
matrix : CUBE100 N: 1000000, NNZ_LowerA: 3.9700E+06						
256	9.06e+08	1.28	6.03e+12	117.084	51471.67	4.361
matrix : MDUAL2 N: 988605, NNZ_LowerA: 2.9357E+06						
32	1.64e+08	1.89	2.51e+11	47.787	5247.15	1.062
64	1.64e+08	1.63	2.52e+11	25.044	10073.15	0.896
128	1.64e+08	1.59	2.54e+11	16.180	15699.65	1.258
256	1.63e+08	1.72	2.47e+11	8.501	29035.15	2.267
matrix : AORTA N: 577771, NNZ_LowerA: 1.7067E+06						
8	7.86e+07	1.65	8.43e+10	60.566	1391.39	1.448
16	7.72e+07	1.92	8.20e+10	33.599	2440.07	0.855
32	7.84e+07	2.33	8.58e+10	26.651	3220.07	0.614
64	7.76e+07	2.08	8.26e+10	11.056	7472.44	0.530
128	7.96e+07	2.11	8.63e+10	7.088	12175.47	0.639
256	7.92e+07	2.03	8.53e+10	3.675	23198.01	1.190
matrix : CUBE65 N: 274625, NNZ_LowerA: 1.0858E+06						
32	1.20e+08	1.18	3.46e+11	49.944	6927.02	0.446
64	1.21e+08	1.16	3.50e+11	27.313	12822.10	0.471
128	1.20e+08	1.13	3.47e+11	14.931	23259.87	0.567
256	1.21e+08	1.20	3.49e+11	8.665	40239.15	1.112



Table 1.6 Performance for factorization and solve phases on Cray T3E-1200 (2)

np	NNZ_L	Imbal	FAC_OPC	Ntime	Nperf	Ttime
matrix : S3DKQ4M2 N: 90449, NNZ_LowerA: 2.4600E+06						
2	1.81e+07	1.00	7.21e+09	14.910	483.39	0.512
4	1.81e+07	1.01	7.24e+09	7.797	928.73	0.278
8	1.81e+07	1.15	7.22e+09	4.444	1625.43	0.163
16	1.82e+07	1.27	7.27e+09	2.632	2761.03	0.116
32	1.82e+07	1.26	7.31e+09	1.432	5106.93	0.093
64	1.82e+07	1.31	7.24e+09	0.859	8420.89	0.109
128	1.79e+07	1.33	7.05e+09	0.517	13625.52	0.090
256	1.74e+07	1.40	6.81e+09	0.361	18877.46	0.114
matrix : COPTER2 N: 55476, NNZ_LowerA: 4.0771E+05						
2	9.54e+06	1.03	5.43e+09	11.271	482.08	0.431
4	9.46e+06	1.14	5.26e+09	5.912	890.14	0.240
8	9.34e+06	1.32	5.16e+09	3.355	1537.90	0.146
16	9.68e+06	1.44	5.57e+09	2.390	2328.89	0.103
32	9.62e+06	1.38	5.46e+09	1.211	4510.29	0.074
64	9.55e+06	1.46	5.37e+09	0.737	7284.67	0.066
128	9.64e+06	1.59	5.45e+09	0.526	10350.46	0.075
256	9.65e+06	1.68	5.55e+09	0.413	13432.96	0.140
matrix : BCSSTK18 N: 11948, NNZ_LowerA: 8.0500E+04						
2	6.18e+05	1.35	1.01e+08	0.540	186.20	0.076
4	6.54e+05	1.88	1.17e+08	0.411	283.40	0.045
8	6.72e+05	2.04	1.22e+08	0.258	473.10	0.029
16	8.10e+05	2.30	1.82e+08	0.220	824.45	0.023
32	7.84e+05	1.75	1.61e+08	0.125	1288.02	0.022
64	7.08e+05	2.14	1.20e+08	0.101	1191.23	0.020
128	6.80e+05	2.44	1.12e+08	0.087	1282.71	0.024

- [4] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. Pspases: Scalable parallel direct solver library for sparse symmetric positive definite linear systems (version 1.0), user's manual. Technical Report 97-059, Department of Computer Science, University of Minnesota, 1997. Available via <http://www.cs.umn.edu/~mjoshi/pspases>.
- [5] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.
- [6] G. Karypis, K. Schlogel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library (version 1.0), user's manual. Technical Report 97-060, Department of Computer Science, University of Minnesota, 1997. Available via <http://www.cs.umn.edu/~metis>.
- [7] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwood City, CA, 1994.
- [8] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal of Matrix Analysis and Applications*, 11:134–172, 1990.
- [9] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.

## 2 PARALLEL FACTORIZATION ALGORITHMS WITH ALGORITHMIC BLOCKING

Jaeyoung Choi

*School of Computing  
Soongsil University  
Seoul 156-743, KOREA  
choi@comp.soongsil.ac.kr*

**Abstract:** LU and QR factorizations are the most widely used method for solving dense linear systems of equations, and have been extensively studied and implemented on vector and parallel computers. Since each parallel computer has very different performance ratios of computation and communication, the optimal computational block sizes, which generate the maximum performance of an algorithm, are different from one another. Therefore, the data matrix must be distributed with the machine specific optimal block size before the computation. Too small or large block size makes getting good performance on a machine near impossible. In such a case, getting a better performance may require a complete redistribution of the data matrix. If the block size is very small or very large, it is impossible to expect good performance on the machine. The data matrix might be completely redistributed to get the better performance.

In this chapter, we present parallel LU and QR factorization algorithms with an “algorithmic blocking” strategy on 2-dimensional block cyclic data distribution. With the algorithmic blocking, is possible obtaining the best performance irrespective of the physical block size. The algorithms are implemented and compared with the ScaLAPACK factorization routines on the Intel Paragon computer.

### 2.1 INTRODUCTION

In many linear algebra algorithms the distribution of work may become uneven as the algorithm proceeds, for example as in LU factorization algorithm [8, 11], in which rows and columns are successively eliminated from the computation. The way in which a matrix is distributed over the processes has a major impact on the load balance and

communication characteristics of a parallel algorithm, and hence largely determines its performance and scalability.

The two-dimensional block cyclic data distribution [9, 14], in which matrix blocks separated by a fixed stride in the row and column directions are assigned to the same process, has been used as a general purpose basic data distribution for parallel linear algebra software libraries because of its scalability and load balance properties. And most of the parallel version of algorithms have been implemented on the two-dimensional block cyclic data distribution [5, 19].

Since each parallel computer has very different performance ratios of computation and communication, the optimal block sizes, which generate the maximum performance of an algorithm, are different from one another. The data matrix must be distributed with the machine specific optimal block size before the computation. Too small or large block size makes getting good performance on a machine near impossible. In such case, getting a better performance may require a complete redistribution of the data matrix.

The matrix multiplication,  $\mathbf{C} \Leftarrow \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$ , might be the most fundamental operation in linear algebra. Several parallel matrix multiplication algorithms have been proposed on the two-dimensional block-cyclic data distribution [1, 6, 10, 13, 18]. The parallel matrix multiplication scheme using a series of rank- $K$  updates [1, 18] has been demonstrated its high performance, scalability, and simplicity. It is assumed that the data matrices are distributed on the two-dimensional block cyclic data distribution and the column block size of  $\mathbf{A}$  and the row block size of  $\mathbf{B}$  are  $K$ . However getting a good performance when the block size is very small or very large is difficult, since the computation are not effectively overlapped with the communication. The LCM concept [10, 17] has been introduced to DIMMA [6] to use a computationally optimal block size irrespective of the physical block size for the parallel matrix multiplication. In DIMMA, if the physical block size is smaller than the optimal block size, the small blocks are combined into a larger block. And if the physical block size is larger than the optimal block size, the block is divided into smaller pieces. This is the “algorithmic blocking” strategy.

There have been several efforts to develop parallel factorization algorithms with the algorithmic blocking on distributed-memory concurrent computers. Lichtenstein and Johnsson [16] developed and implemented block-cyclic order elimination algorithms for LU and QR factorization on the Connection Machine CM-200. They used a cyclic order elimination on block data distribution, only scheme that the Connection Machine system compilers supported.

And P. Bangalore [3] has tried to develop a data distribution-independent LU factorization algorithm. He recomposed computational panels to obtain a computationally optimal block size, but followed the original matrix ordering. According to the results, the performance is superior to the other case, in which the matrix is redistributed when the block size is very small. He used a tree-type communication scheme to make computational panels from several columns of processes. However, if possible, using a pipelined communication scheme, which overlaps communication and computation effectively, would be more efficient.

The actual algorithm which is selected at runtime depending on input data and machine parameters is called “polyalgorithms” [4]. We are developing “PoLAPACK” (Poly LAPACK) factorization routines, in which computers select the optimal block size at run time according to machine characteristics and size of data matrix. In this paper, we expanded and generalized the idea in [16]. We developed and implemented parallel LU and QR factorization algorithms with the algorithmic blocking strategy on 2-dimensional block cyclic data distribution. With PoLAPACK, it is possible to have the best performance irrespective of the physical data-distribution on distributed-memory concurrent computers if all of the processes have the same size of matrices. The PoLAPACK factorization routines doesn’t follow the conventional computational ordering, and the solution vector  $x$  (possible to expand the vector  $x$  to the solution matrix  $X$ ) which was computed with optimal block size may be recomposed to the same position as the right-hand sided vector,  $b$ .

The PoLAPACK LU and QR factorization routines are implemented based on ScaLAPACK, but the internals are very different. ScaLAPACK uses global parameters, but PoLAPACK uses both global and local parameters because of the computational complexity to compute indices, which represent the current row and column of processes, and the global size of the matrix and local sizes of submatrices to be computed. Currently the PoLAPACK factorization routines are implemented based on block cyclic data distribution, but it is also easy to apply the idea to any matrices which are decomposed with other decompositions.

The PoLAPACK LU and QR factorization algorithms are implemented on Intel Paragon computer at Samsung Advanced Institute of Technology, Suwon, Korea, and their performance is compared with the corresponding ScaLAPACK factorization routines.

## 2.2 POLAPACK LU FACTORIZATION ALGORITHM

We will briefly explain the right-looking version of the block LU factorization algorithm, whose parallel implementation in distributed-memory concurrent computers minimizes communication and distributes the computation among processes, therefore it has good load balance and scalability on distributed-memory concurrent computers [11]. The LU factorization applies a sequence of Gaussian eliminations to form  $P \cdot A = L \cdot U$ , where  $A$  and  $L$  are  $M \times N$  matrices, and  $U$  is an  $N \times N$  matrix.  $L$  is a unit lower triangular matrix,  $U$  is an upper triangular matrix, and  $P$  is a permutation matrix, which is stored in a  $\min(M, N)$  vector.

At the  $k$ -th step of the computation ( $k = 1, 2, \dots$ ), it is assumed that the  $m \times n$  submatrix of  $A(k)$  ( $m = M - (k - 1)n_b$ ,  $n = N - (k - 1)n_b$ ) is to be partitioned as follows,

$$\begin{aligned} P \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \\ &= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix} \end{aligned}$$

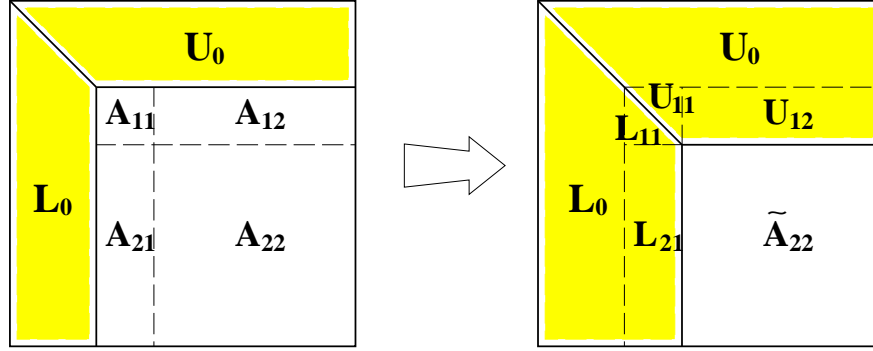


Figure 2.1 A snapshot of block LU factorization.

where the block  $A_{11}$  is  $n_b \times n_b$ ,  $A_{12}$  is  $n_b \times (n - n_b)$ ,  $A_{21}$  is  $(m - n_b) \times n_b$ , and  $A_{22}$  is  $(m - n_b) \times (n - n_b)$ .  $L_{11}$  is a unit lower triangular matrix, and  $U_{11}$  is an upper triangular matrix. At first, a sequence of Gaussian eliminations is performed on the first  $m \times n_b$  panel of  $A(k)$  (i.e.,  $A_{11}$  and  $A_{21}$ ). Once this is completed, the matrices  $L_{11}$ ,  $L_{21}$ , and  $U_{11}$  are known, and we can rearrange the block equations,

$$\begin{aligned} U_{12} &\leftarrow (L_{11})^{-1} A_{12}, \\ A(k+1) &\leftarrow A_{22} - L_{21} U_{12} = L_{22} U_{22}. \end{aligned}$$

The block LU factorization can be done by recursively applying the steps outlined above to the  $(m - n_b) \times (n - n_b)$  matrix of  $A(k+1)$ . Figure 2.1 shows a snapshot of the block LU factorization. It shows how the column panel,  $L_{11}$  and  $L_{21}$ , and the row panel,  $U_{11}$  and  $U_{12}$ , are computed, and how the trailing submatrix  $A_{22}$  is updated. In the figure, the shaded areas represent data for which the corresponding computations are complete. Later, row interchanges will be applied to  $L_0$  and  $L_{21}$ .

In the ScaLAPACK factorization routines [8], a column of processes performs a factorization on its own column of blocks, and broadcasts it to others. Then all of processes update the rest of the data matrix. The basic unit of the computation is the physical size of the block, with which the data matrix is already distributed over processes.

The basic LU factorization routine is to find the solution vector  $x$  after applying LU factorization to  $A$  from the following linear equation

$$A x = b.$$

That is, after converting  $A$  to  $P \cdot A = L \cdot U$ , then compute  $y$  from the following equation

$$L y = b_0, \tag{2.1}$$

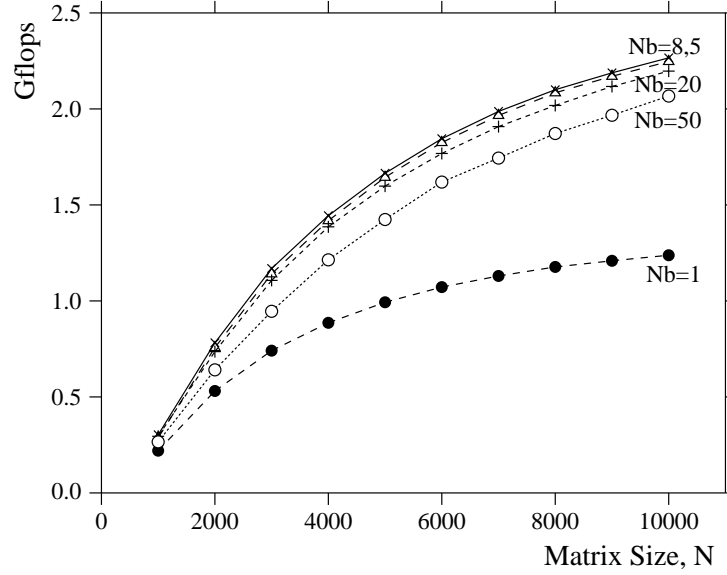


Figure 2.2 Performance of ScaLAPACK LU factorization routine on an  $8 \times 8$  Intel Paragon

which is defined from  $LU \cdot x = P \cdot b = b_0$  and  $U \cdot x = y$ . Now it is possible to compute  $x$  from the following equation

$$U x = y. \quad (2.2)$$

Most of LU factorization algorithms including LAPACK [2] and ScaLAPACK find the solution vector  $x$  after computing the factorization of  $P \cdot A = L \cdot U$ . We measured the performance the ScaLAPACK LU factorization routine and its solution routine with various block sizes on the Intel Paragon. Figure 2.2 shows the performance on an  $8 \times 8$  process grid from  $N = 1,000$  to  $10,000$  with block sizes of  $N_b = 1, 5, 8, 20$ , and  $50$ . It shows that the best performance is obtained when  $N_b = 8$ , and almost the same but slightly slower when  $N_b = 5$ . The performance deteriorated to 2-3% when  $N_b = 20$ , 13% when  $N_b = 50$ , and more than 45% when  $N_b = 1$ . If the data matrix is distributed with  $N_b = 1$ , it may be much more efficient to perform the factorization after redistributing the data matrix with the optimal block size of  $N'_b = 8$ .

In ScaLAPACK, the performance of the algorithm is greatly affected by the block size. However the PoLAPACK LU factorization is implemented with the concept of algorithmic blocking and always shows the best performance of  $N_b = 8$  irrespective of physical block sizes.

If a data matrix  $A$  is decomposed over 2-dimensional  $p \times q$  processes with block cyclic data distribution, it may be possible to regard the matrix  $A$  being decomposed along the row and column directions of processes. Then the new decomposition along the row and column directions are the same as applying permutation matrices from the left and the right, respectively. One step further, if we want to compute a matrix

with a new block size, we may need to redistribute the matrix, and we can assume that the redistributed matrix is of the form  $P_p \cdot A \cdot P_q^T$ , where  $P_p$  and  $P_q$  are permutation matrices. And it may be possible to avoid redistributing the matrix physically if the new computation doesn't follow the given ordering of the matrix  $A$ . That is, by assuming that the given matrix  $A$  is redistributed with a new block size and the resulting matrix is  $P_p \cdot A \cdot P_q^T$ , it is now possible to apply the factorization to  $A$  with a new optimal block size for the computation. And this factorization will show the same performance regardless of the physical block sizes if each process gets the same size of the submatrix of  $A$ . These statements are illustrated with the following equations,

$$(P_p A P_q^T) \cdot (P_q x) = P_p \cdot b. \quad (2.3)$$

Let  $A_1 = P_p A P_q^T$ , and  $x_1 = P_q x$ . After factorizing  $A_1$  to  $P_1 A_1 = P_1 \cdot (P_p A P_q^T) = L_1 \cdot U_1$ , then we compute the solution vector  $x$ . The above equation Eq. 2.3 is transformed as follows:

$$L_1 \cdot U_1 \cdot (P_q x) = L_1 \cdot U_1 \cdot x_1 = P_1 \cdot (P_p b) = b_1.$$

As in Eq. 2.1 and Eq. 2.2,  $y_1$  is computed from

$$L_1 \cdot y_1 = b_1, \quad (2.4)$$

and  $x_1$  is computed from

$$U_1 \cdot x_1 = y_1. \quad (2.5)$$

Finally the solution vector  $x$  is computed from

$$P_q \cdot x = x_1. \quad (2.6)$$

The computations are performed with  $A$  and  $b$  in place with the optimal block size, and  $x$  is computed with  $P_q$  as in Eq. 2.6. But we want  $P_p \cdot x$  rather than  $x$  in order to make  $x$  have the same physical data distribution as  $b$ . That is, we compute

$$P_p \cdot x = P_p \cdot P_q^T \cdot x_1. \quad (2.7)$$

Assume that we have a  $24 \times 24$  block matrix  $A$  is distributed with a block size of  $N_b$  over a  $2 \times 3$  process grid as in Figure 2.3. When the optimal block size is the same as the physical block size (i.e.,  $N'_b = N_b$ ), the computational ordering of the PoLAPACK is the same as that of the ScaLAPACK as in Figure 2.3(a). However, if the optimal block size is twice the physical block size (i.e.,  $N'_b = 2 \cdot N_b$ ), the PoLAPACK routine computes with two columns of blocks,  $A(:, 0)$  and  $A(:, 3)$ , and two rows of blocks,  $A(0, :)$  and  $A(2, :)$  at the first step as in Figure 2.3(b). But those two columns and rows of blocks belong to the same column



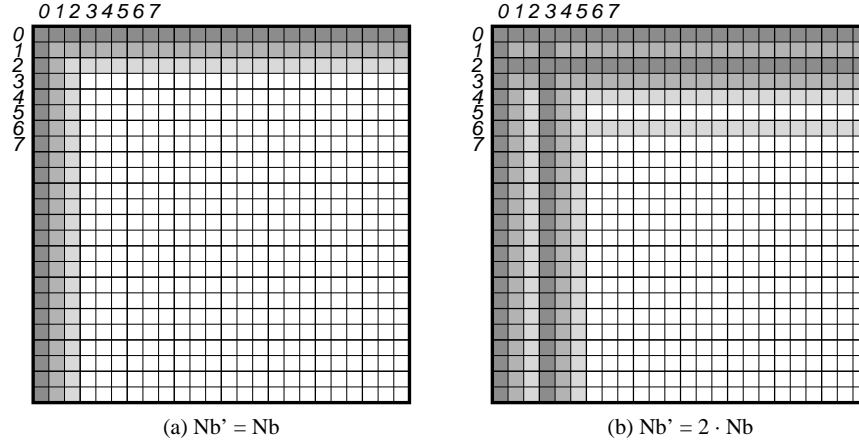


Figure 2.3 Computation procedures in progress in PoLAPACK. (a) when the optimal block size is the same as the physical block size, (b) when the optimal block size is twice the physical block size.

and row of processes, respectively. In this example, the computational orderings of the row and column blocks have been changed  $(0, 2), (1, 3), (4, 6), (5, 7), \dots$ , and  $(0, 3), (1, 4), (2, 5), (6, 9), (7, 10), \dots$ , and the new orderings are obtained by multiplying  $P_p$  to  $A$  from the left and  $P_q^T$  from the right, respectively. This procedure doesn't include any physical data redistribution.

## 2.3 IMPLEMENTATION OF POLAPACK LU FACTORIZATION

The PoLAPACK LU factorization routine is composed of three parts.

(1) LU Factorization: We implemented the right-looking version of the LU factorization algorithm as in Section 2.2. We implemented the LAPACK LU factorization routines, DGETRF, DGETF2, DLAMAX, DLAPIV, DLASWP, and DGETRS to the corresponding PoLAPACK LU factorization routines, PoDGETRF, PoDGETF2, PoDLAMAX, PoDLAPIV, PoDLASWP, and PoDGETRS, respectively. And we partially implemented the Level 2 and Level 3 BLAS routines, DGER, DGEMM, and DTRSM, to the corresponding PolyBLAS routines, PoDGER, PoD-GEMM, and PoDTRSM, respectively, for the PoLAPACK LU factorization.

Figure 2.4 shows the computational procedure of the PoLAPACK LU factorization. Assume that a matrix  $A$  of  $12 \times 12$  blocks is distributed over a  $2 \times 3$  process grid as in Figure 2.4(a), and the LU routine computes 2 blocks at a time (imagine  $N_b = 4$  but  $N'_b = 8$ ). Since the routine follows 2-D block cyclic ordering, the positions of the diagonal blocks are regularly changed by incrementing one column and one row of processes at each step. However, if  $A$  is  $9 \times 9$  blocks as in Figure 2.4(b), the next diagonal block of  $A(5, 6)$  on  $p_{(3)}$  is  $A(7, 7)$  on  $p_{(4)}$ , not on  $p_{(1)}$ . Then the next block is

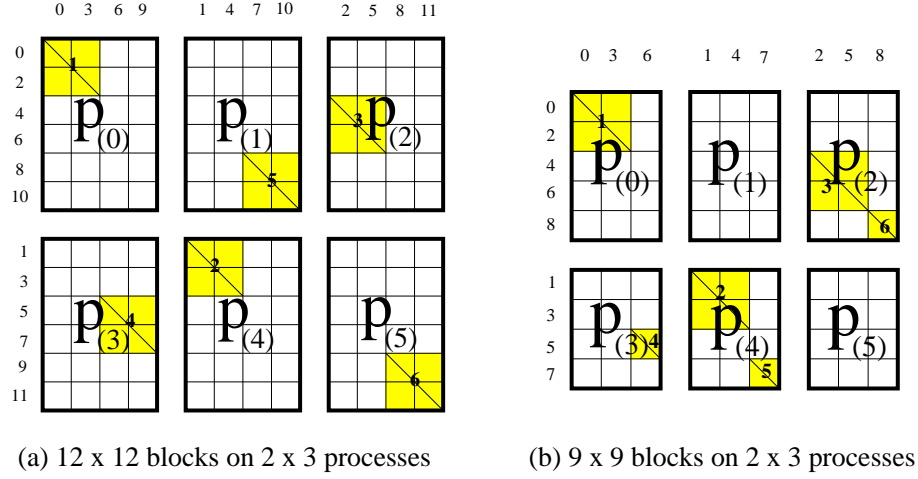


Figure 2.4 Computational Procedure in PoLAPACK. Matrices of  $12 \times 12$  and  $9 \times 9$  blocks are distributed on  $2 \times 3$  processes with  $N_b$  and  $N'_b = 2 \cdot N_b$ , respectively.

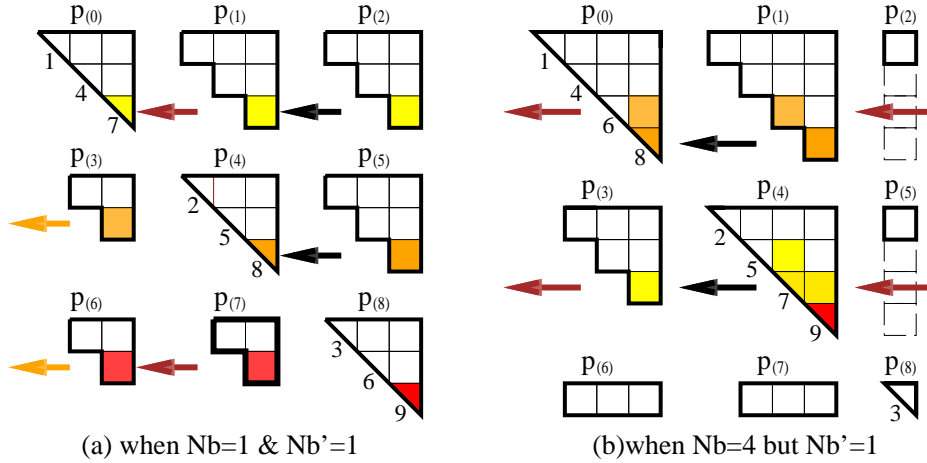


Figure 2.5 A snapshot of PoLAPACK solver. A matrix  $T$  of  $9 \times 9$  blocks is distributed on  $3 \times 3$  processes with  $N_b = 1$  and  $N'_b = 4$ , respectively, while the optimal computational block size for both cases is  $N'_b = 1$ .

$A(8, 8)$  on  $p_{(2)}$ . The computational procedure of the PoLAPACK is very complicated, and it is difficult to implement.

(2) Triangular Solver: We implemented the Li and Coleman's algorithm [15] on a two dimensional process grid in the PolyBLAS routine, PoDTRSM. The 2-dimensional version of the algorithm is already implemented in the PBLAS routine, PDTRSM [7].

But the implementation of PoDTRSM is much more complicated since the diagonal block may not be located regularly if  $p$  is not equal to  $q$  as in Figure 2.4.

Even if  $p$  is equal to  $q$ , the implementation is still complicated. Figure 2.5(a) shows a snapshot of the Li and Coleman's algorithm [15] from processes point-of-view, where  $9 \times 9$  blocks of an upper triangular matrix  $T$  are distributed over a  $3 \times 3$  process grid with  $N_b = N'_b = 1$ . Let's look over the details of the algorithm to solve  $x = T \setminus b$ .

At first, the last block at  $p_{(8)}$  computes  $x(9)$  from  $T(9, 9)$  and  $b(9)$ . Processes in the last column update 2 blocks - actually  $p_{(2)}$  and  $p_{(5)}$  update  $b(7)$  and  $b(8)$ , respectively - and send them to their left processes. The rest of  $b$  ( $b(1 : 6)$ ) are updated later. At the second step,  $p_{(4)}$  computes  $x(8)$  from  $T(8, 8)$  and  $b(8)$ , the latter is received from  $p_{(5)}$ . While  $p_{(1)}$  receives  $b(7)$  from  $p_{(2)}$ , updates it, and sends it to  $p_{(0)}$ ,  $p_{(7)}$  updates a temporal  $b(6)$  and sends it to  $p_{(6)}$ .

Figure 2.5(b) shows the same size of the matrix distribution  $T$  with  $N_b = 4$ , but it is assumed that the matrix  $T$  is derived with an optimal block size  $N'_b = 1$ . So the solution routine has to solve the triangular equations of Eq. 2.4 and Eq. 2.5 with  $N'_b = 1$ . The first two rows and the first two columns of processes have 4 rows and 4 columns of  $T$ , respectively, while the last row and the last column have 1 row and 1 column, respectively. Since  $N'_b = 1$ , the computation starts from  $p_{(4)}$ , which computes  $x(9)$ . Then  $p_{(1)}$  and  $p_{(4)}$  update  $b(8)$  and  $b(7)$ , respectively, and send them to their left. The rest of  $b$  ( $b(1 : 6)$ ) are updated later. At the next step,  $p_{(0)}$  computes  $x(8)$  from  $T(8, 8)$  and  $b(8)$ , the latter is received from  $p_{(1)}$ . While  $p_{(3)}$  receives  $b(7)$  from  $p_{(4)}$ , updates it, and sends it to the left  $p_{(5)}$ ,  $p_{(0)}$  updates a temporal  $b(6)$  and sends it to its left  $p_{(2)}$ . However  $p_{(2)}$  and  $p_{(5)}$  don't have their own data to update or compute at the current step, and hand them over to their left without touching the data. The PoLAPACK solver has to comply with this kind of all the abnormal cases.

(3) Solution Vector Redistribution: It may be needed to redistribute the solution vector  $x$  with  $P_p \cdot P_q^T \cdot x$  as in Eq. 2.7. However, if  $p$  is equal to  $q$ , then  $P_p$  becomes  $P_q$ , and  $P_p \cdot P_q^T \cdot x = x$ , therefore, the redistribution is not necessary. But if  $p$  is not equal to  $q$ , the redistribution of  $x$  is required to get the solution with the same data distribution as the right hand vector  $b$ . And if  $p$  and  $q$  are relatively prime, then the problem is changed to all-to-all personalized communication. We are currently implementing the routine and the details will be discussed in a separate paper.

We implemented the PoLAPACK LU factorization routine and measured its performance on an  $8 \times 8$  process grid. Figure 2.6 shows the performance of the routine with the physical block sizes of  $N_b = 1, 5, 8, 20$ , and  $50$ , but the optimal block size of  $N'_b = 8$ . As shown in Figure 2.6, the performance lines are very close to the others and always show the near maximum performance irrespective of the physical block sizes. Since all processes don't have the same size of the submatrices of  $A$  with various block sizes in some cases, some processes have more data to compute than others, which causes computational load imbalance among processes and the resulting performance degradation. For example, the line with a small white circle in Figure 2.6 shows the case of  $N_b = 50$ , in which processes in the first half have more data to compute than processes in the second half if the matrix size  $N = 7,000$  or  $9,000$ .

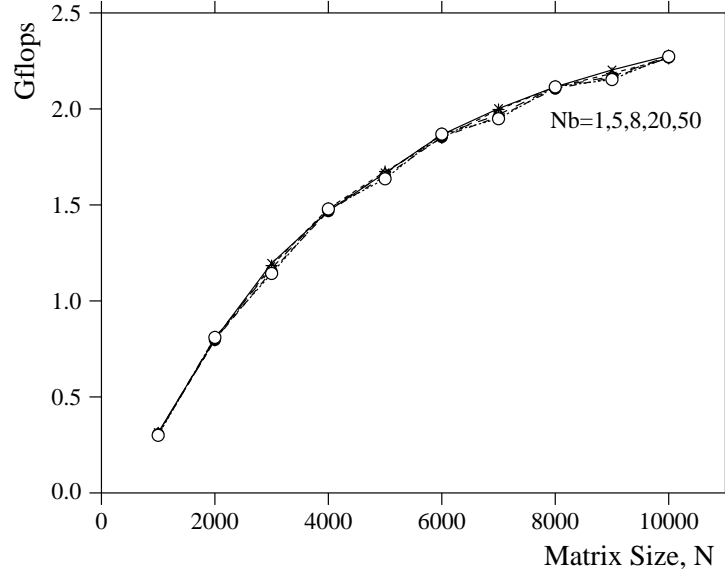


Figure 2.6 Performance of PoLAPACK LU factorization routine on an  $8 \times 8$  Intel Paragon

## 2.4 POLAPACK QR FACTORIZATION

The QR factorization is used to solve the least squares problem [12],

$$\min_{x \in \mathbb{R}} \|Ax - b\|_2. \quad (2.8)$$

Given an  $M \times N$  matrix  $A$ , the QR factorization is computed by  $A = QR$ , where  $Q$  is an  $M \times M$  orthogonal matrix, and  $R$  is an  $M \times N$  upper triangular matrix.

The PoLAPACK QR factorization and its solution of the factored matrix equations are performed in a manner analogous to the PoLAPACK LU factorization and the solution of the triangular systems. Eq. 2.8 can be changed as

$$\|Ax - b\|_2 = \|(P_p A P_q^T) \cdot (P_q x) - P_p b\|_2$$

where  $P_p$  and  $P_q$  are permutation matrices. Let  $A_2 = P_p \cdot A \cdot P_q^T$ ,  $x_2 = P_q x$ , and  $b_2 = P_p b$ . After factorizing  $A_2 = P_p A P_q^T$  to  $Q_2 \cdot R_2$ , then we compute the solution vector  $x$ . The above equation is transformed as follows:

$$\|Q_2 \cdot R_2 \cdot (P_q x) - (P_p b)\|_2 = \|Q_2 \cdot R_2 \cdot x_2 - b_2\|_2. \quad (2.9)$$

If applying  $Q_2^T$  to  $b_2$  to form  $b_3$ , Eq. 2.9 is changed as

$$\|R_2 \cdot x_2 - Q_2^T b_2\|_2 = \|R_2 \cdot x_2 - b_3\|_2,$$

where  $b_3 = Q_2^T b_2$ . Then  $x_2$  is computed from

$$R_2 \cdot x_2 = b_3.$$

Finally the solution vector  $x$  is computed from

$$P_q \cdot x = x_2.$$

Again as in Eq. 2.7, we want  $P_p \cdot x = P_p P_q^T x$  rather than  $x$  to make  $x$  have the same physical data distribution as  $b$ .

Figures 2.7 and 2.8 show the performance of the ScaLAPACK and PoLAPACK QR factorizations and their solution routines, respectively, with  $N_b = 1, 6, 8, 20$  and 50 on an  $8 \times 8$  process grid of the Intel Paragon. Performance of the ScaLAPACK QR factorization algorithm depends on the physical block size, and the best performance is obtained when  $N_b = 6$ . However the PoLAPACK QR factorization algorithm, which computes with the optimal block size of  $N'_b = 6$ , always shows the near maximum performance independent of physical block sizes.

## 2.5 CONCLUSIONS

Generally in other parallel factorization algorithms, a column of processes performs the factorization on a column of blocks of  $A$  at a time, whose block size is already fixed, and then the other processes update the rest of the matrix. If the block size is very small or very large, then the processes can't give their optimal performance, and the data matrix may be redistributed for a better performance. The computation follows the original ordering of the matrix.

It may be faster and more efficient to perform the computation, if possible, by combining several columns of blocks if the block size is small, or by splitting a large column of blocks if the block size is large. This is the main concept of the algorithmic blocking. The PoLAPACK factorization routines rearrange the ordering of the computation. They compute  $P_p A P_q^T$  instead of directly computing  $A$ . They proceed the computation with the optimal block size without physically redistributing  $A$ . And the solution vector  $x$  is computed by solving triangular systems, then converting  $x$  to  $P_p P_q^T x$ . The final rearrangement of the solution vector can be omitted if  $p = q$ .

According to the results of the ScaLAPACK and the PoLAPACK LU and QR factorization routines on the Intel Paragon in Figures 2.2, 2.6, 2.7 and 2.8, the ScaLAPACK factorizations have a large performance difference with different values of  $N_b$ , but the PoLAPACK factorizations always show a steady performance, which is near the best, irrespective of the values of  $N_b$ . The algorithms we presented in this paper are developed based on the block cyclic data distribution. This simple idea can be easily applied to the other data distributions. But it is required to develop specific algorithms to rearrange the solution vector for each distribution.

Currently we are implementing and testing the redistribution of the solution vector to support the case of  $p \neq q$ . Future work will focus on developing PoLAPACK Cholesky factorization routine. Since the Cholesky factorization handles a symmetric triangular matrix, it is impossible to implement PoLAPACK Cholesky factorization with the algorithmic blocking technique on process grids of  $p \neq q$ . However it may be

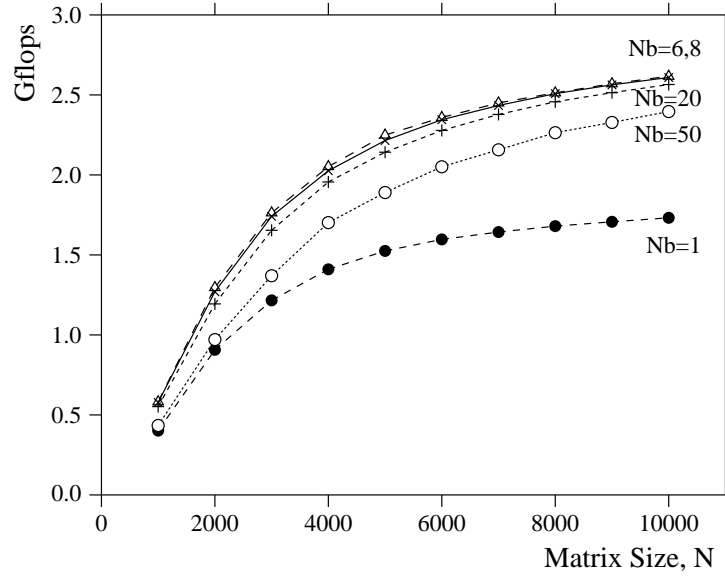


Figure 2.7 Performance of ScaLAPACK QR factorization routine on an  $8 \times 8$  Intel Paragon

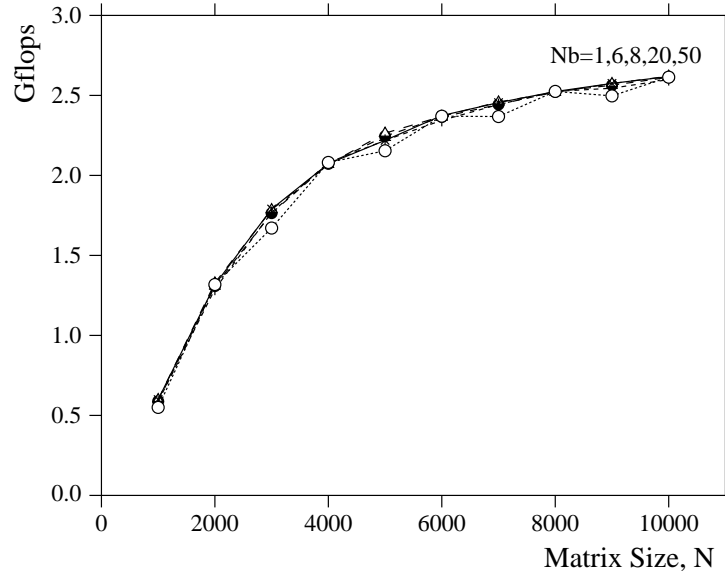


Figure 2.8 Performance of PoLAPACK QR factorization routine on an  $8 \times 8$  Intel Paragon

possible to implement it on process grids of  $p = q$ . If  $p \neq q$ , the PoLAPACK Cholesky computes the factorization with the physical block size. It is possible to get the benefit of the algorithmic blocking on the Cholesky factorization for the limited case of  $p = q$ .

And we intend to supply the complete version of the PoLAPACK in the near future, which includes the three factorization routines and supports all numeric data types.

## References

- [1] Agarwal, R. C., Gustavson, F. G., and Zubair, M. (1994). A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681.
- [2] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1990). LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press.
- [3] Bangalore, P. V. (1995). The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master Thesis, Mississippi State University.
- [4] Blackford, L., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. (1997a). ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of SIAM Conference on Parallel Processing*.
- [5] Blackford, L., Choi, J., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. (1997b). *ScaLAPACK Users' Guide*. SIAM Press, Philadelphia, PA.
- [6] Choi, J. (1998). A New Parallel Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *Concurrency: Practice and Experience*, 10:655–670.
- [7] Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A. P., Walker, D. W., and Whaley, R. C. (1995). A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. LAPACK Working Note 100, Technical Report CS-95-292, University of Tennessee.
- [8] Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A. P., Walker, D. W., and Whaley, R. C. (1996). The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184.
- [9] Choi, J., Dongarra, J. J., and Walker, D. W. (1992). The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*, pages 3–15. Elsevier Science Publishers.
- [10] Choi, J., Dongarra, J. J., and Walker, D. W. (1994). PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570.
- [11] Dongarra, J. J. and Ostrouchov, S. (1990). LAPACK Block Factorization Algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee.

- [12] Golub, G. H. and Loan, C. V. V. (1989). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD. Second Edition.
- [13] Huss-Lederman, S., Jacobson, E. M., Tsao, A., and Zhang, G. (1994). Matrix Multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, 6:571–594.
- [14] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.
- [15] Li, G. and Coleman, T. F. (1986). A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM J. of Sci. Stat. Computing*, 9:485–502.
- [16] Lichtenstein, W. and Johnsson, S. L. (1993). Block-Cyclic Dense Linear Algebra. *SIAM J. of Sci. Stat. Computing*, 14(6):1259–1288.
- [17] Petitet, A. (1996). Algorithmic Redistribution Methods for Block Cyclic Decompositions. Ph.D. Thesis, University of Tennessee, Knoxville.
- [18] van de Geijn, R. and Watts, J. (1995). SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, Technical Report CS-95-286, University of Tennessee.
- [19] van de Geijn, R. A. (1997). *Using PLAPACK*. The MIT Press, Cambridge.



### 3 PARALLEL SOLUTION OF STIFF ORDINARY DIFFERENTIAL EQUATIONS

Thomas Rauber,

*Institut für Informatik  
Universität Halle-Wittenberg  
Kurt-Mothes-Str.1  
06120 Halle (Saale), Germany  
rauber@informatik.uni-halle.de*

Gudula Rünger

*Institut für Informatik  
Universität Leipzig  
Augustusplatz 10-11  
04109 Leipzig, Germany  
ruenger@informatik.uni-leipzig.de*

**Abstract:** We consider the parallel solution of stiff ordinary differential equations with implicit Runge-Kutta (RK) methods. In particular, we describe parallel implementations of classical implicit RK methods and parallel implementations of diagonal-implicitly iterated RK (DIIRK) methods which have been especially developed for a parallel execution. DIIRK methods have computational redundancy but provide an additional source of parallelism in the form of independent nonlinear equation systems to be solved in each time step. To compare the runtimes of the parallel RK methods, we apply them to systems of ordinary differential equations resulting from a spatial discretization of partial differential equations. As programming platform, we use a Cray T3E.

### 3.1 INTRODUCTION

Implicit methods are used for the solution of stiff initial value problems (IVPs) for ordinary differential equations (ODEs) of the form  $y'(t) = f(t, y(t))$  with  $y(t_0) = y_0$  for a given initial vector  $y_0$ . The right-hand side  $f$  is usually a non-linear function. Popular one-step methods include implicit Runge-Kutta (RK) methods, diagonally implicit RK methods, Rosenbrock-type methods, and extrapolation methods. We focus on implicit RK methods which result in safe and precise codes for medium precision [3]. Modern RK methods use an error control and stepsize selection mechanism so that the accuracy of the resulting discrete solution is consistent with a predefined error tolerance.

Large systems of ODEs arise, e.g., from a spatial discretization of time-dependent partial differential equations (PDEs). The solution of those systems may lead to large computation times which motivates a parallel implementation. Approximations to successive points in time have to be computed one after another. Therefore, parallel solution methods for ODEs concentrate on the parallel execution of one time step. Several parallel methods have been proposed, including methods with computational redundancy. Such methods require more evaluations of the function  $f$  than the best sequential methods but provide more parallelism instead, because several computations within one time step can be computed in parallel. Combining the available degree of parallelism of these methods with a distribution of the evaluation of the components of  $f$  across several processors results in solution methods with good scalability properties, i.e., large relative speedup values can be reached, especially if the evaluation of  $f$  is costly.

In this article, we compare the parallel execution of implicit RK methods with a parallel execution of methods that have been especially designed for a parallel execution to provide a good scalability. In particular, we consider diagonal-implicitly iterated RK (DIIRK) methods which are obtained by iterating the stage-vector system of an  $s$ -stage implicit RK method for a fixed number of times and by introducing an additional diagonal matrix [8, 9]. By choosing the number  $m$  of iterations of the stage-vector system as  $m = r - 1$  where  $r$  is the order of the original implicit RK method, an implicit method of the same order  $r$  results. A specific characteristic of the DIIRK method is that for each iteration of the stage-vector system, the non-linear equation system of size  $s \cdot n$  (where  $n$  is the size of the ODE system) actually consists of  $s$  decoupled subsystems of size  $n$  each. Thus, one time step of the DIIRK method requires the solution of  $s \cdot m$  non-linear systems of size  $n$ , leading to a computational complexity of  $s \cdot m \cdot n^3$ , if a Newton method is used and if the Newton iterations are based on a direct solution method for the internal linear equation systems. For the original implicit RK method, the non-linear equation system is not decoupled, thus the computational complexity of one time step is  $s^3 \cdot n^3$ .

For a parallel execution, the main difference between the implicit RK methods and the DIIRK methods is that the  $s$  non-linear equation systems of one DIIRK step are independent of each other giving the possibility for a task parallel execution. The processors are partitioned into  $s$  disjoint groups and each processor group is responsible for the solution of one of the  $s$  equation systems. Since each nonlinear equation system

requires the same amount of work, the processor groups are chosen to be of equal size. The communication behavior of the parallel implementations is mainly determined by the internal communications of the Newton method which uses single-transfer, broadcast, and global reduction operations.

For an implementation as message-passing program on a specific parallel machine, the communication behavior of the machine and the characteristics of the specific ODE system have a large influence on the resulting performance. As target machines, we consider a Cray T3E. As example systems, we consider two classes of ODE systems that differ in the evaluation time of  $f$ .

The rest of the article is organized as follows. Section 3.2 gives a short description of the implicit RK methods that are considered in this paper. Section 3.3 describes parallel implementations for the different methods. Section 3.4 applies the parallel implementations to example problems with different characteristics. Section 3.5 concludes.

## 3.2 IMPLICIT RUNGE–KUTTA METHODS

In this section, we give a short description of implicit Runge–Kutta methods and describe how the DIIRK methods are derived. In the subsequent subsections, we sketch an implementation, show how the number of function evaluations can be reduced for the DIIRK methods and summarize the stepsize control mechanism using embedded solutions.

### 3.2.1 Runge–Kutta methods

Runge–Kutta (RK) methods are one–step solution methods for IVPs of ODEs [2] [3] [7]. The time-steps of an implicit RK method compute iteration vectors  $\mathbf{y}_\kappa$ ,  $\kappa = 1, 2, 3, \dots$  one after another according to the formula

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^s b_l \mathbf{f}(\mathbf{v}_l) \quad (3.1)$$

where the stage vectors  $\mathbf{v}_l$ ,  $l = 1, \dots, s$ , are defined by the  $s \cdot n$  dimensional fully implicit system:

$$\mathbf{v}_l = \mathbf{y}_\kappa + h \sum_{i=1}^s a_{li} \mathbf{f}(\mathbf{v}_i) \quad l = 1, \dots, s. \quad (3.2)$$

The  $s$ –dimensional vectors  $\mathbf{b} = (b_1, \dots, b_s)$  and  $\mathbf{c} = (c_1, \dots, c_s)$  and the  $s \times s$  matrix  $\mathbf{A} = (a_{li})_{l,i=1,\dots,s}$  describe the basic RK method. The number  $s$  is called the *stage* of the RK method and  $h$  is the stepsize. The formulae (3.1) and (3.2) are given in *stage-value* notation where the function  $\mathbf{f}$  is applied only to stage vectors  $(\mathbf{v}_1, \dots, \mathbf{v}_s)$  (in contrast to a notation where  $\mathbf{f}$  is applied to the entire right hand side of equations (3.1) and (3.2)). The stage-value notation is more convenient for the derivation of

iterated methods. The iteration vector  $\mathbf{y}_\kappa$  represents the approximation of the solution  $\mathbf{y}$  at time  $t_\kappa$ , i.e.,  $\mathbf{y}_\kappa = \tilde{\mathbf{y}}(t_\kappa)$ . The computation of  $\mathbf{y}_{\kappa+1}$  from  $\mathbf{y}_\kappa$  is called a *time step*. (We use the convention to set vectors, e.g.,  $\mathbf{y}_\kappa$ ,  $\mathbf{v}_l$ , in bold type.)

For the computation of the stage vectors  $\mathbf{v}_l$ ,  $l = 1, \dots, s$ , according to Equation (3.2), an implicit system of size  $s \cdot n$  has to be solved where  $n$  is the size of the ODE system. The system can be described by  $F(\mathbf{z}) = 0$  using a function  $F : \mathbb{R}^{s \cdot n} \rightarrow \mathbb{R}^{s \cdot n}$  where  $\mathbf{z} \in \mathbb{R}^{s \cdot n}$  is the concatenation of the  $s$  stage vectors solving Equation (3.2), i.e.,  $\mathbf{z} = (\mathbf{z}_1, \dots, \mathbf{z}_s)$ .  $F$  is composed of  $s$  components, i.e.,  $F = (F_1, \dots, F_s)$  with  $F_l : \mathbb{R}^{s \cdot n} \rightarrow \mathbb{R}^n$  defined by

$$F_l(\mathbf{z}_1, \dots, \mathbf{z}_s) = \mathbf{z}_l - \mathbf{y}_\kappa - h \sum_{i=1}^s a_{li} \mathbf{f}(\mathbf{z}_i) \quad (3.3)$$

Popular implicit RK methods are the  $s$ -stage Radau Ia and Radau IIa methods of order  $2s - 1$  and the Lobatto IIIA, IIIB, and IIIC methods of order  $2s - 2$  [3].

### 3.2.2 DIIRK methods

Iterated RK methods are obtained by iterating Equation (3.2) for a fixed number of times [8]. Iterated RK methods are explicit methods that are suitable for the solution of nonstiff systems of ODEs. For the construction of the (implicit) DIIRK method, an additional diagonal matrix  $\mathbf{D}$  of dimension  $s \times s$  is introduced in Equation (3.2) resulting in the system:

$$\mathbf{v}_l = \mathbf{y}_\kappa + h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i) + h d_{ll} \mathbf{f}(\mathbf{v}_l) \quad (3.4)$$

for  $l = 1, \dots, s$ . One time step of the DIIRK method consists of a fixed number  $m$  of iteration steps of Equation (3.4) using the term  $h d_{ll} \mathbf{f}(\mathbf{v}_l)$  as implicit part. The initial iteration vector is provided by the predictor method. Here we use a simple one-step predictor method, the *last-step-value* predictor (see [9]), which yields the following standard computation scheme for the DIIRK method:

$$\mathbf{v}_l^{(0)} = \mathbf{y}_\kappa \quad l = 1, \dots, s, \quad (3.5)$$

$$\mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) + h d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \quad (3.6)$$

$$l = 1, \dots, s, \quad j = 1, \dots, m,$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^s b_l \mathbf{f}(\mathbf{v}_l^{(m)}). \quad (3.7)$$

One time step from  $\kappa$  to  $\kappa + 1$  according to system (3.5), (3.6), (3.7) is called a *macrostep* of the DIIRK method. The execution of one iteration step from  $j$  to  $j + 1$  of Equation (3.6) is called a *corrector step*. The number  $m$  of corrector steps determines

the convergence order of the method. The convergence order of the DIIRK method is  $r^* = \min(r, m + 1)$  where  $r$  is the order of the implicit basic RK method [8].

Considering all  $m$  corrector iterations of one macro step, the DIIRK method is equivalent to a diagonal-implicitly RK method with block structure. This can be illustrated by the Butcher array of the method [9]:

$$\begin{array}{c|ccccccc}
 j=0 & \mathbf{O} & & & & & & \\
 j=1 & \mathbf{A} - \mathbf{D} & \mathbf{D} & & & & & \\
 j=2 & \mathbf{O} & \mathbf{A} - \mathbf{D} & \mathbf{D} & & & & \\
 j=3 & \mathbf{O} & \mathbf{O} & \mathbf{A} - \mathbf{D} & \mathbf{D} & & & \\
 \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \\
 j=m & \mathbf{O} & \dots & & & \mathbf{O} & \mathbf{A} - \mathbf{D} & \mathbf{D} \\
 \hline
 & \mathbf{0}^T & \dots & & & \mathbf{0}^T & \mathbf{b}^T & 
 \end{array}$$

where  $\mathbf{A}$  is the matrix of the RK corrector,  $\mathbf{D}$  is the diagonal matrix,  $\mathbf{O}$  is the  $s \times s$  matrix containing 0 in every entry and  $\mathbf{0}^T$  is the  $s$  dimensional vector containing 0.

### 3.2.3 Computing the stage vectors

The main part of the computation time of the implicit RK methods and the DIIRK methods is used for the computation of the stage vectors  $\mathbf{v}_1^{(j)}, \dots, \mathbf{v}_s^{(j)}$  which requires to solve nonlinear implicit systems according to Equations (3.2) and (3.6), respectively. Implicit nonlinear systems of equations arising in implicit Runge-Kutta methods are usually solved by an iteration method such as the Newton method [4] which we use in our implementation.

The DIIRK method has several properties which are considered to produce a fast (parallel) implementation:

- An automatic stepsize control is possible without additional computational effort as the iterations of the RK method in the corrector steps provide embedded solutions [3], see Section 3.2.5.
- Each of the  $m$  systems (3.6) of size  $s \cdot n$  actually consists of  $s$  decoupled subsystems of size  $n$ , each of which specifies one iteration vector  $\mathbf{v}_l^{(j)}$ ,  $l = 1, \dots, s$ . For the computation of  $\mathbf{v}_l^{(j)}$  we have to solve the system  $F_{j,l}(\mathbf{z}) = 0$  with  $F_{j,l} : \mathbb{R}^n \rightarrow \mathbb{R}^n$  defined as follows

$$F_{j,l}(\mathbf{z}) = \mathbf{z} - \mathbf{y}_\kappa - h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) - h d_{li} \mathbf{f}(\mathbf{z}) \quad (3.8)$$

- In each corrector step the number of function evaluations of  $\mathbf{f}$  can be reduced by performing some precomputations in the previous corrector step. The precomputed function values can also be used for the update step (3.7) and the stepsize

control such that both can be implemented without any additional function evaluations.

In the following two subsections, we describe two of those properties in more detail, the reduction of the number of function evaluation in Subsection 3.2.4 and the stepsize control mechanism in Subsection 3.2.5.

### 3.2.4 Reduced Number of Function Evaluations

The number of function evaluations in the corrector step  $j + 1$  for the computation of  $\mathbf{v}_l^{(j+1)}$ ,  $l = 1, \dots, s$ , can be reduced by exploiting the corrector step  $j$ . By a reformulation of Equation (3.6) of corrector step  $j$  we get:

$$\mathbf{f}(\mathbf{v}_l^{(j)}) = \left( \mathbf{v}_l^{(j)} - \mathbf{y}_\kappa - h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) \right) / (hd_{ll}) \quad (3.9)$$

for  $l = 1, \dots, s$  which represents an alternative way for computing  $\mathbf{f}$  at argument vector  $\mathbf{v}_l^{(j)}$ . All vectors used on the right hand side of Equation (3.9) are known from corrector step  $j$ , i.e., we can compute the values of  $\mathbf{fval}_l^{(j)} = \mathbf{f}(\mathbf{v}_l^{(j)})$  for  $l = 1, \dots, s$  immediately after the corrector step  $j$  has been finished. Instead of (3.8), we now use

$$F_{j,l}(\mathbf{z}) = \mathbf{z} - \mathbf{y}_\kappa - h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{fval}_i^{(j-1)} - hd_{ll} \mathbf{f}(\mathbf{z}). \quad (3.10)$$

The computation of  $\mathbf{f}(\mathbf{v}_l^{(j)})$  according to formula (3.9) does not only save computation time but also avoids an increase of the approximation error that arises when applying  $\mathbf{f}$  to  $\mathbf{v}_l^{(j)}$ .

The computation scheme for one macrostep of the DIIRK method with the reduced number of function evaluations has the form:

$$\mathbf{fval}_l^{(0)} = \mathbf{f}(\mathbf{y}_\kappa) \quad l = 1, \dots, s \quad (3.11)$$

$$\mathbf{w}_l^{(j)} = h \sum_{i=1}^s (a_{li} - d_{li}) \mathbf{fval}_i^{(j-1)} \quad (3.12)$$

$$\mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + \mathbf{w}_l^{(j)} + hd_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \quad (3.13)$$

$$\mathbf{fval}_l^{(j)} = \left( \mathbf{v}_l^{(j)} - \mathbf{y}_\kappa - \mathbf{w}_l^{(j)} \right) / (hd_{ll}), \quad (3.14)$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^s b_l \mathbf{fval}_l^{(m)}. \quad (3.15)$$

for  $l = 1, \dots, s$  and  $j = 1, \dots, m$ .

### 3.2.5 Stepsize Control

One-step methods for the solution of ODE systems in an interval  $t_0 \leq t \leq t_{end}$  perform several macrosteps to approximate the solution  $\mathbf{y}$  at the points  $t_0, t_1, t_2, \dots, t_{end}$  where

$t_{\kappa+1} = t_{\kappa} + h_{\kappa}$ . In order to achieve a good solution and to maintain a fast computation time, the stepsizes  $h_{\kappa}$  have to be chosen as large as possible while guaranteeing small approximation errors.

For the problem of selecting appropriate stepsizes, we adopt an automatic stepsize control from [2] which uses two different approximations  $\mathbf{y}_{\kappa+1}$  and  $\tilde{\mathbf{y}}_{\kappa+1}$  for the solution  $\mathbf{y}(t_{\kappa+1})$  computed with the same stepsize  $h$ . The approximation vector  $\mathbf{y}_{\kappa+1}$  is accepted if  $error \leq bound$ , where  $error = \|\mathbf{y}_{\kappa+1} - \tilde{\mathbf{y}}_{\kappa+1}\|$  and  $bound = \max(|\mathbf{y}_{\kappa}|, |\mathbf{y}_{\kappa+1}|)$ . In this case  $h_{new}$  is used to compute  $\mathbf{y}_{\kappa+2}$  with:

$$h_{new} = h * \min(6, \max(\frac{1}{3}, 0.9 * \left(\frac{bound}{error}\right)^{1/(ord+1)})) \quad (3.16)$$

where  $ord$  is the minimal convergence order of the approximation method used. If  $error > bound$ , the computation of  $\mathbf{y}_{\kappa+1}$  is rejected and is repeated with stepsize  $h_{new}$ .

The DIIRK method provides several approximation solution when using the vectors  $\mathbf{v}_l^{(j)}$ ,  $l = 1, \dots, s$ , for a specific  $j$ ,  $j < m$ , and Equation (3.7)

$$\mathbf{y}^{(j)} = \mathbf{y}_{\kappa} + h \sum_{l=1}^s b_l \mathbf{f}(\mathbf{v}_l^{(j)}).$$

The vectors  $\mathbf{y}^{(j)}$  represent embedded solutions of successively increasing order  $\min(r, j+1)$  where  $r$  is the order of the basic implicit RK method [8], [2], [5]. Most stepsize control mechanisms use solutions  $\mathbf{y}^{(j)}$  for which the order of  $\mathbf{y}_{\kappa+1}$  and  $\mathbf{y}^{(j)}$  differ by 1. Therefore, we choose  $j = m - 1$ . The error is computed according to the formula:

$$\begin{aligned} error &= \|\mathbf{y}_{\kappa+1} - \mathbf{y}^{(m-1)}\| \\ &= |h| * \left\| \sum_{l=1}^s b_l * \left( \mathbf{f}(\mathbf{v}_l^{(m)}) - \mathbf{f}(\mathbf{v}_l^{(m-1)}) \right) \right\|. \end{aligned} \quad (3.17)$$

We can also use the precomputed function evaluations for the error computation:

$$error = |h| * \left\| \sum_{l=1}^s b_l * (\mathbf{fval}_l^{(m)} - \mathbf{fval}_l^{(m-1)}) \right\|. \quad (3.18)$$

### 3.3 PARALLEL IMPLEMENTATIONS

In this section, we give a short overview of parallel implementations for the methods from Section 3.2.

#### 3.3.1 Parallel implicit RK methods

For a parallel implementation, the time steps of an  $s$ -stage implicit RK method are executed one after another. Each time step consists of the computation of the  $s$

stage vectors  $\mathbf{v}_1, \dots, \mathbf{v}_s$  by a Newton method and the computation of the iteration vector  $\mathbf{y}_{k+1}$  according to Equation (3.1). The main computational effort lies in the Newton method to solve the system  $F(\mathbf{z}) = 0$ , see Equation (3.3). The Newton method performs several iterations that have to be computed one after another. In each iteration step, the entries of the Jacobi matrix

$$DF(\mathbf{z}^{(k)}) = \left( \frac{\partial F_i}{\partial \mathbf{z}_j}(\mathbf{z}^{(k)}) \right)_{i,j=1,\dots,n}$$

at  $\mathbf{z}^{(k)}$  have to be computed where  $\mathbf{z}^{(k)} \in \mathbb{R}^{s \cdot s}$  is the current approximation of the concatenated stage vectors. The entry of  $DF$  in row  $i$  and column  $j$  is computed by a forward difference approximation, i.e.,

$$\frac{\partial F_i}{\partial \mathbf{z}_j}(\mathbf{z}^{(k)}) = \frac{1}{r_j} (F_i(\mathbf{z}^{(k)} + r_j \mathbf{e}_j) - F_i(\mathbf{z}^{(k)})) \quad (3.19)$$

where  $\mathbf{e}_j \in \mathbb{R}^n$  is the  $j$ th column of the unit matrix and  $r_j \in \mathbb{R}$  is a suitable interval.  $F_i : \mathbb{R}^{s \cdot s} \rightarrow \mathbb{R}$  is the  $i$ th component of  $F$ . The linear equation system

$$DF(\mathbf{z}^{(k)})\mathbf{y}^{(k)} = -F(\mathbf{z}^{(k)})$$

is solved to compute the correction vector  $\mathbf{y}^{(k)}$ . We use by a Gaussian elimination to find  $\mathbf{y}^{(k)}$ , since a direct method does not require the Jacobi matrix to fulfill a specific condition. The approximation vector  $\mathbf{z}^{(k)}$  is updated by

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} + \mathbf{y}^{(k)}.$$

To obtain a good load balance for the Gaussian elimination, we use a cyclic distribution of the rows of the coefficient matrix among the processors. Correspondingly, we partition the computation of the Jacobi matrix in such a way among the processors that each processor  $q$  computes all rows  $i$  with  $q = i \bmod p$  where  $p$  is the number of executing processors. If the approximation vector of the Newton iteration is held replicated, the computation of the Jacobi matrix can be performed in a distributed way without communication. Using the forward difference approximation, the computation of one row of the Jacobi matrix requires the evaluation of  $s \cdot n + 1$  components of  $F$ . If one component has evaluation time  $T_F$ , the computation of the Jacobi matrix requires time

$$T_1(n \cdot s, p) = \frac{s \cdot n}{p} (s \cdot n + 1) \cdot T_F.$$

The Gaussian elimination requires several communication operations in each step:

- Using column pivoting, a global accumulation operation with a maximum reduction has to be performed to determine the pivot element.
- A single-transfer operation has to be used to exchange the current row with the pivot row.
- A broadcast operation is used to make the pivot row available to all processors.



Moreover, in each step of the backward substitution, a broadcast operation is used to make the newly computed element available to all processors. This makes each element of the solution vector available to each processor. Thus, the approximation vector of the Newton iteration can be held replicated. The replicated generation of the solution vector of the Gaussian elimination requires only local operations to produce the updated approximation vector in a replicated way.

Since the Newton iteration delivers the stage vectors replicated, the iteration vector  $\mathbf{y}_{k+1}$  can be computed block-wise, i.e., each processor computes a contiguous block of  $n/p$  components of  $\mathbf{y}_{k+1}$ . After this computation,  $\mathbf{y}_{k+1}$  is made available to each processor by a multi-broadcast operation to which each processor contributes the  $n/p$  components of  $\mathbf{y}_{k+1}$  that it has locally computed. This operation is necessary, since each processor needs the entire vector  $\mathbf{y}_{k+1}$  in the next time step.

The parallel implementation of implicit RK methods essentially consists of a parallel execution of the Newton method which requires nearly all of the execution time of a time step. In each time step, a nonlinear system with  $s \cdot n$  components has to be solved where the solution vector is the concatenation of the approximations to the stage vectors. Thus, the approximations of the stage vectors are computed simultaneously in contrast to the computation of the stage vectors in explicit RK methods, see, e.g., [1], where the stage vectors have to be computed one after another. Increasing the number of stages leads to an increase in the size of the nonlinear equation system and, hence, to an increase in the available degree of parallelism. The available degree of parallelism is not limited by data dependencies within one time step. But there is of course a dependence between successive time steps. Moreover, the execution time of one time step is quite large compared to the execution time of the multi-broadcast operation at the end of each time step, i.e., the efficiency is mainly dominated by the efficiency of the Newton iteration.

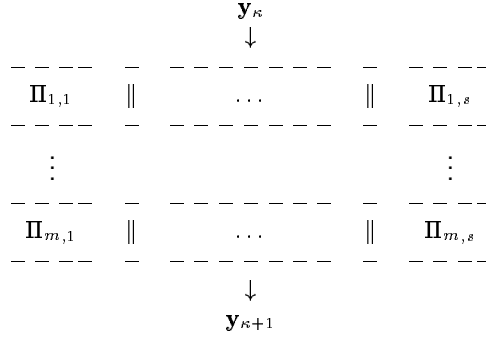
### 3.3.2 Parallel DIIRK methods

In each corrector step of the DIIRK method, we have to solve  $s$  independent, nonlinear subsystems each of size  $n$  instead of one system of size  $s \cdot n$ . The existence of independent subsystems not only decreases the computational effort but can also be exploited for a parallel implementation. We compute the stage vectors  $\mathbf{v}_l^{(j)}$ ,  $l = 1, \dots, s$ , by solving the subsystems by a separate Newton iteration. Let  $\Pi_{j,l}$  for  $l = 1, \dots, s$  and  $j = 1, \dots, m$  denote the subsystems of one corrector step  $j$ .  $\Pi_{j,l}$  is the nonlinear system  $F_{j,l}(\mathbf{z}) = 0$  with  $F_{j,l}$  according to Equations (3.8) or (3.10), respectively. The following figure illustrates the order in which the systems  $\Pi_{j,l}$  have

RK method	stages	order	DIIRK iterations
Radau IA	2	3	2
Radau IIA	3	5	4
Lobatto IIIC	5	8	7

Figure 3.1 Characteristics of implicit RK methods.

to be solved because of data dependencies:



The symbol  $\parallel$  indicates independent computations, i.e.,  $\Pi_{1,1} \parallel \dots \parallel \Pi_{1,s}$  are independent of each other and can be solved in parallel. The horizontal dashed lines indicate a synchronization point and data exchange.

In the following, we consider two execution schemes

- a *consecutive* execution scheme in which the systems  $\Pi_{j,1}, \dots, \Pi_{j,s}$  of one corrector step  $j$  are solved one after another by all processors available and
- a *group* execution scheme in which these systems are solved concurrently by  $s$  independent groups of processors.

The consecutive execution scheme has a similar computation and communication behavior than the parallel implementation of the implicit RK method described in Section 3.3.1. The difference is that the systems to be solved for the DIIRK method are of size  $n$  whereas the systems to be solved for the implicit RK methods are of size  $s \cdot n$ . For the group implementation, the set of processors is divided into  $s$  groups  $G_1, \dots, G_s$ . Group  $G_l$  contains about the same number  $g_l = \lceil p/s \rceil$  or  $g_l = \lfloor p/s \rfloor$  of processors. In each corrector iteration step  $j = 1, \dots, m$ , group  $G_l$  is responsible for the computation of one subvector  $\mathbf{v}_l^{(j)}$ ,  $l \in \{1, \dots, s\}$ . The solution of the nonlinear systems is done with a group implementation of the Newton method, i.e., the communication operations are replaced by group communication operations.

### 3.4 RUNTIME TESTS

For the RK methods in Figure 3.1, we compare the runtime of implicit RK methods and DIIRK methods of the same order  $r$ . For the DIIRK methods, the number of

iterations is chosen as  $m = r - 1$ . As target machine we consider a Cray T3E which we program with C and MPI (Message Passing Interface). We performed runtime tests with  $p = 4, 8, 16$ , and 32 processors. As test problems, we consider two ODE systems: an ODE system that results from a spatial discretization of a two-dimensional reaction-diffusion equation [3] and an ODE system that results from a Fourier-Galerkin approximation of a Schrödinger–Poisson system for the numerical simulation of a collisionless electron plasma [6]. The 2D Brusselator equation is described by

$$\begin{aligned}\frac{\partial u}{\partial t} &= 1 + u^2 v - 4.4u + \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\ \frac{\partial v}{\partial t} &= 3.4u - u^2 v + \alpha \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right)\end{aligned}$$

for  $0 \leq x \leq 1$ ,  $0 \leq y \leq 1$ ,  $t \geq 0$ . The unknown functions  $u$  and  $v$  describe the concentrations of the two substances. A Neumann boundary condition

$$\frac{\partial u}{\partial n} = 0, \quad \frac{\partial v}{\partial n} = 0,$$

and the initial conditions

$$u(x, y, 0) = 0.5 + y, \quad v(x, y, 0) = 1 + 5x$$

are used. A standard discretization of the spatial derivatives with a uniform grid with mesh size  $1/(N - 1)$  leads to the following ODE system of dimension  $2N^2$ :

$$\begin{aligned}\frac{du_{ij}}{dt} &= 1 + u_{ij}^2 v_{ij} - 4.4u_{ij} + \alpha(N-1)^2 (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}) \\ \frac{dv_{ij}}{dt} &= 3.4u_{ij} - u_{ij}^2 v_{ij} + \alpha(N-1)^2 (v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{i,j}).\end{aligned}$$

For  $\alpha = 0.02$ , this ODE system is stiff [3]. The Schrödinger–Poisson system for the numerical simulation of a collisionless electron plasma [6] is described by

$$\begin{aligned}i\hbar \frac{\partial \Psi}{\partial t} &= -\frac{\hbar}{2m} \frac{\partial^2 \Psi}{\partial x^2} + e\Phi\Psi \\ \frac{\partial^2 \Phi}{\partial x^2} &= \frac{e n_0}{\varepsilon_0} (1 - \Psi\Psi^*)\end{aligned}$$

with electron charge  $e$  and electron mass  $m$ .  $\Psi$  is the unknown Schrödinger wave function,  $\Phi$  is a potential. A Galerkin approximation with the Galerkin Ansatz

$$\Psi_n = \sum_{|l| \leq n} \alpha_l(t) h_l, \quad \Phi_n = \sum_{|l| \leq n} \beta_l(t) h_l, \quad n \in \mathbf{N},$$

results in the following dimensionless ODE system for the coefficients  $\alpha_l(t)$  and  $\beta_l(t)$  of  $\Psi_n$  and  $\Phi_n$

$$\begin{aligned}i \frac{d\alpha_l}{dt} &= -\lambda_l \alpha_l + \sum_{|j| \leq n} \alpha_j \cdot (\phi_n h_j, h_l), \quad l = 0, \pm 1, \dots, \pm n \\ \beta_l &= \frac{1}{\lambda_l} (|\psi_n|^2, h_l), \quad l = \pm 1, \dots, \pm n\end{aligned}$$

with  $\lambda_l = (\frac{2\pi l}{L})^2$ . The functions

$$h_l(x) = \frac{1}{\sqrt{l}} \exp\left(i \frac{2\pi l}{L} x\right), \quad l \in \mathbb{Z}$$

are an orthonormal basis of  $L^2([0, L])$  with the usual scalar product  $(u, v)$ .

A characteristic of the Brusselator ODE system is that the evaluation time of one component of  $f$  does not depend on the size of the ODE system (*sparse*  $f$ ). Thus, the evaluation time of the complete function  $f$  depends linearly on the number of equations in the ODE system. A characteristic of the Schrödinger ODE system is that the evaluation time for one component of  $f$  depends linearly on the size of the ODE system (*dense*  $f$ ), i.e., the evaluation time of the complete function  $f$  depends quadratically on the number of equations in the ODE system.

To compare the parallel execution time of implicit RK methods and DIIRK methods, we apply the methods from Figure 3.1 to the test problems with different discretizations. Figures 3.2 and 3.3 show the runtimes for methods of order 3 based on the Radau Ia method. Figures 3.4 and 3.5 show the runtimes for methods of order 5 based on the Radau IIa method. Figure 3.6 shows the runtimes for methods of order 8 based on the Lobatto IIIC method. The figures use a logarithmic scale because of the large difference in the execution time for systems of different sizes. The following observations can be made from the runtime tests:

- For small systems, the implicit RK methods lead to smaller execution times than the DIIRK methods in most cases. For larger systems, the DIIRK methods are usually faster than the implicit RK methods.
- For larger systems, the data parallel execution of the DIIRK method is usually slightly faster than the task parallel execution. For smaller systems, the task parallel execution is often faster.

These observations can be made for the Brusselator as well as for the Schrödinger example. For a selection of the method with the fastest parallel execution time, the following rough rules can be used:

- For the Brusselator equation use the implicit RK methods for systems of small sizes and use the data parallel DIIRK method for large systems.
- For the Schrödinger equation, use the implicit RK methods for small systems, the task parallel DIIRK methods for systems of medium size, and the data parallel DIIRK methods for large systems.

### 3.5 CONCLUSIONS

Implicit RK methods with  $s$  stages require the solution of a nonlinear equation system of size  $s \cdot n$  in each time step. DIIRK methods that are based on these methods require the solution of  $m \cdot s$  nonlinear equation system in each time step, each system having size  $n$ . For  $m = r - 1$  where  $r$  is the order of the underlying implicit RK method, a DIIRK method of the same order results. An  $s$ -stage Radau IIa method has for example order  $r = 2s - 1$ . The solution of a nonlinear system of size  $n$  by a Newton

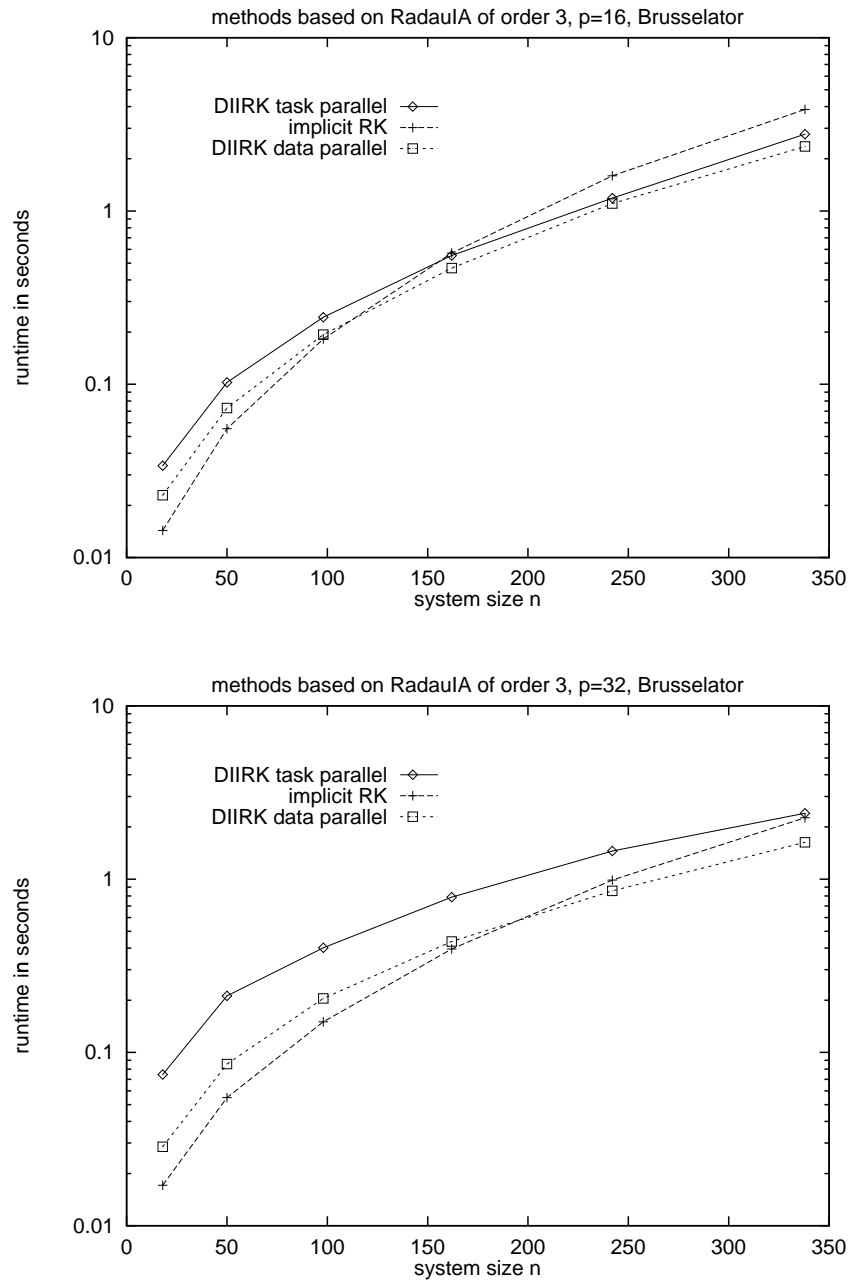


Figure 3.2 Runtimes of methods of order 3 based on RadaulA solving the Brusselator ODE on a T3E using 16 (top) and 32 (bottom) processors.

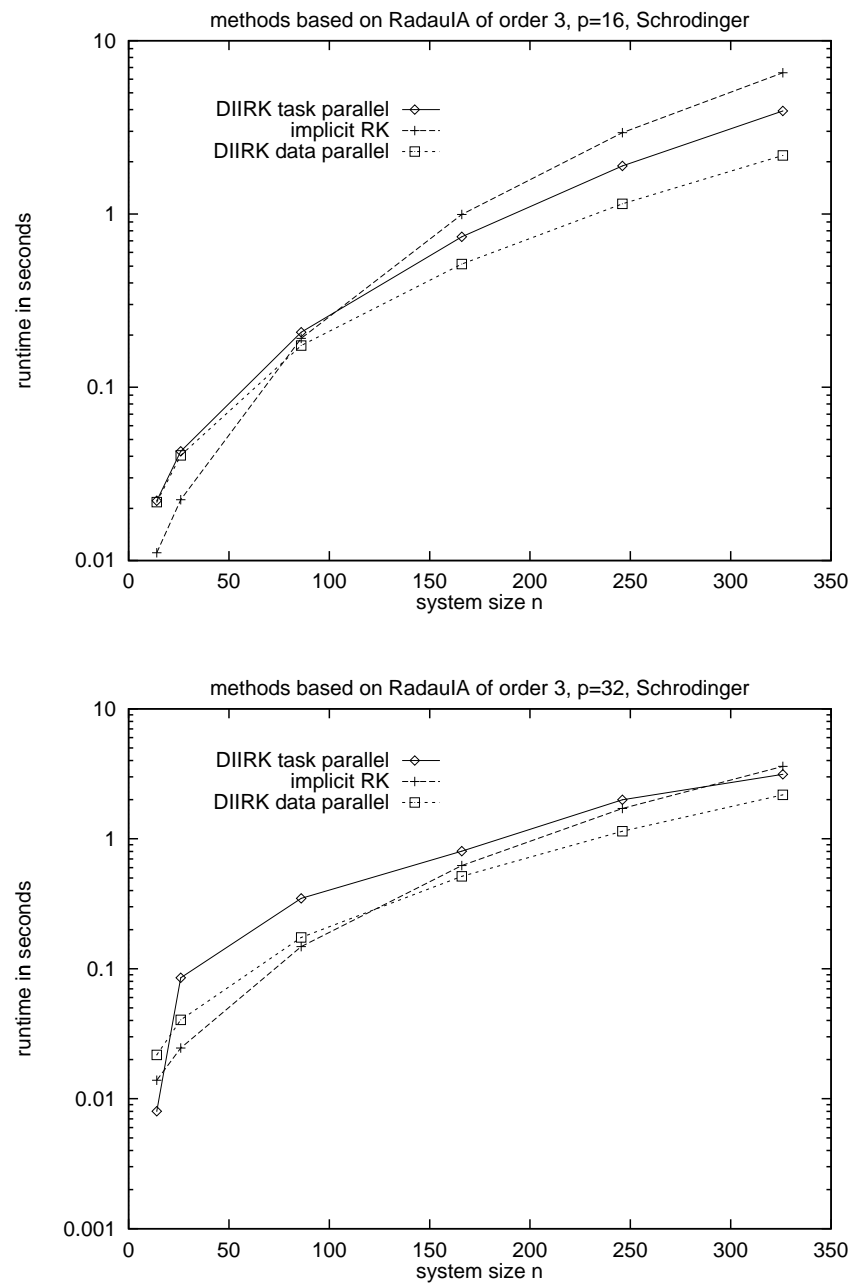


Figure 3.3 Runtimes of methods of order 3 based on Radau IA solving the Schrödinger ODE on a T3E using 16 (top) and 32 (bottom) processors.

iteration requires in each iteration step the computation of a Jacobi matrix and the solution of a linear equation system. Using a forward difference approximation,  $n$  evaluations of the complete function  $F$  with  $n$  components (according to Equations (3.3) or (3.8)) are required for the computation of the Jacobi matrix. The solution of the linear equation systems of size  $n$  with a direct method has runtime  $O(n^3)$ . Thus, the implicit RK methods require the evaluation of  $s^2 \cdot n^2$  components of the right hand side  $f$  of the ODE system and use runtime  $O(s^3 \cdot n^3)$  for the linear equation system in each iteration step of the Newton method. The DIIRK methods on the other hand require the evaluation of  $s \cdot m \cdot n^2$  components of  $f$  and use runtime  $O(s \cdot m \cdot n^3)$  for the solution of linear equation systems. Therefore, for the Radau IIA methods, the number of function evaluations required for one time step of the DIIRK method is larger. However, the additional potential parallelism outweighs this computational disadvantage.

The comparison of the parallel runtimes of implicit RK methods and DIIRK methods depends strongly on the evaluation time of the right hand side  $f$  of the ODE system considered. For the Brusselator ODE system, the evaluation time of one component of  $f$  is constant. Hence, for larger systems, the execution time for one Newton iteration is dominated by the time for the solution of the linear equation systems. For the Schrödinger ODE system, the evaluation time of one component of  $f$  increases linearly with  $n$ , i.e., for larger systems, the computation of the Jacobi matrix has a significant influence on the execution time.

### 3.6 ACKNOWLEDGEMENTS

We would like to thank the Höchstleistungsrechenzentrum of the KFA Jülich, Germany, for giving us access to their Cray T3E.

### References

- [1] S. Bergmann, T. Rauber, and G. Rünger. Parallel Execution of Embedded Runge-Kutta Methods. In *Proc. ParCo'97, to appear*, 1997.
- [2] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin, 1993.
- [3] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II*. Springer, 1991.
- [4] W. Liniger and R.A. Willoughby. Efficient Integration Methods for Stiff Systems of Ordinary Differential Equations. *SIAM Journal of Applied Analysis*, 7:47–66, 1970.
- [5] P.J. Prince and J.R. Dormand. High order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7(1):67–75, 1981.
- [6] T. Rauber and G. Rünger. Parallel Solution of a Schrödinger–Poisson system. In *International Conference on High-Performance Computing and Networking*, pages 697–702. Springer LNCS 919, 1995.

- [7] L.F. Shampine, H.A. Watts, and S.M. Davenport. Solving Nonstiff Ordinary Differential Equations – the State of the Art. *SIAM Review*, 18(3):376–411, 1976.
- [8] P.J. van der Houwen and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.
- [9] P.J. van der Houwen, B.P. Sommeijer, and W. Couzy. Embedded Diagonally Implicit Runge–Kutta Algorithms on Parallel Computers. *Mathematics of Computation*, 58(197):135–159, January 1992.



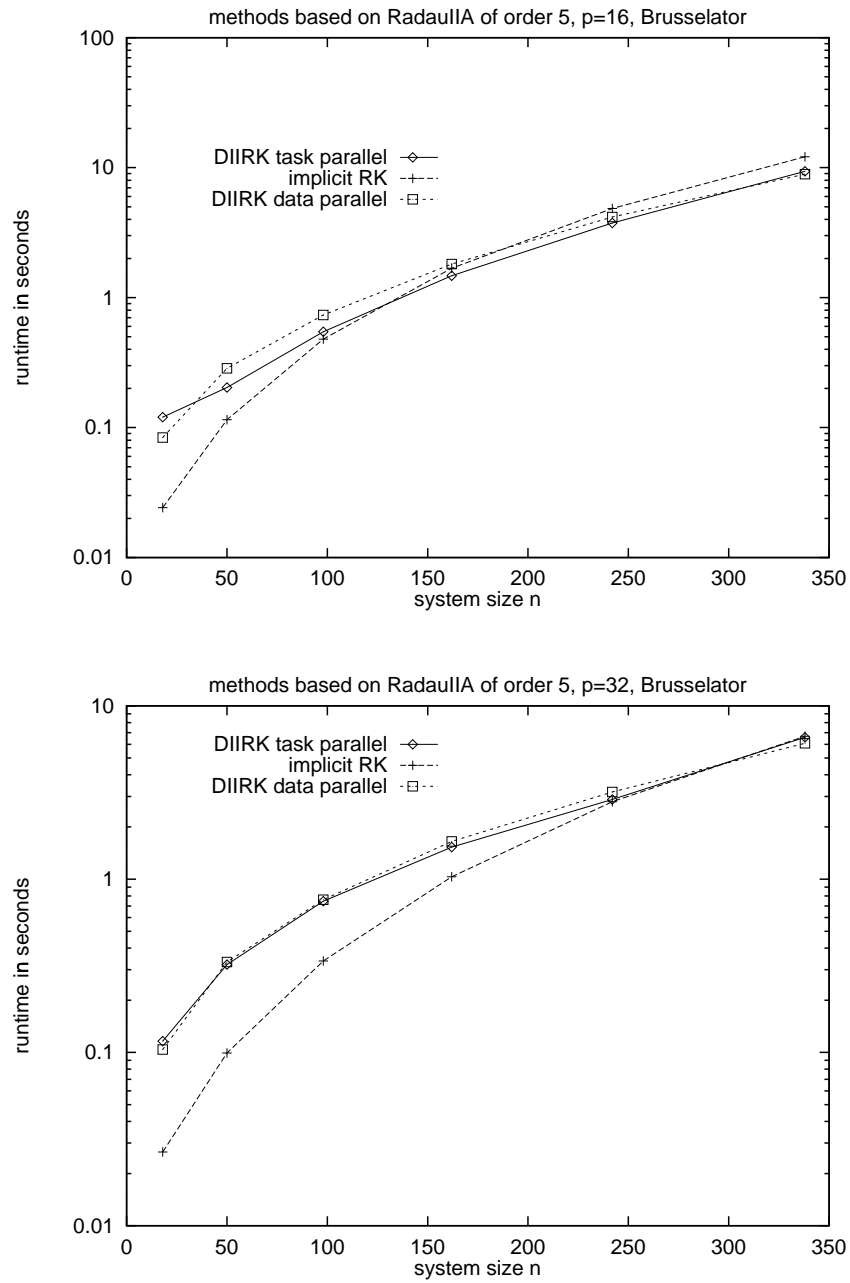


Figure 3.4 Runtimes of methods of order 5 based on Radau IIA solving the Brusselator ODE on a T3E using 16 (top) and 32 (bottom) processors.

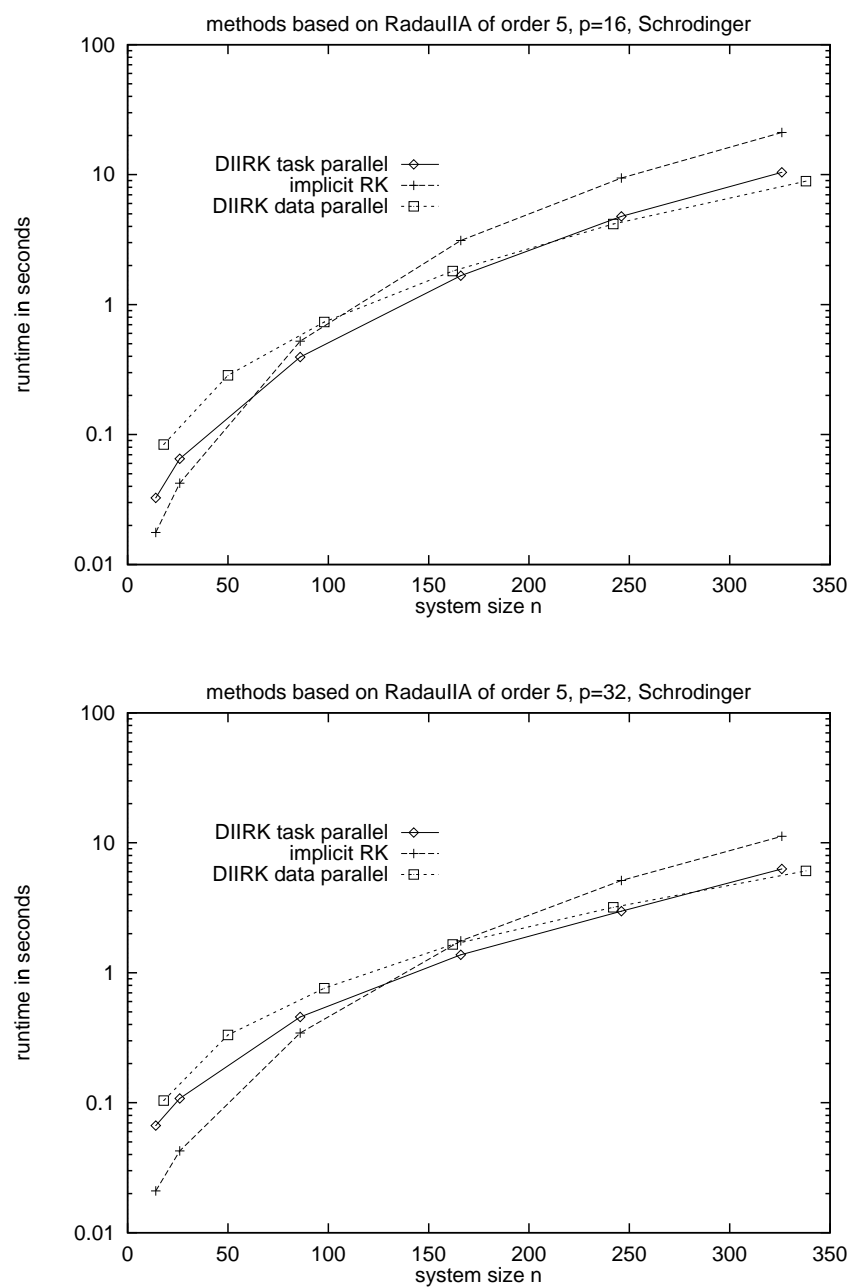


Figure 3.5 Runtimes of methods of order 5 based on Radau IIA solving the Schrödinger ODE on a T3E using 16 (top) and 32 (bottom) processors.

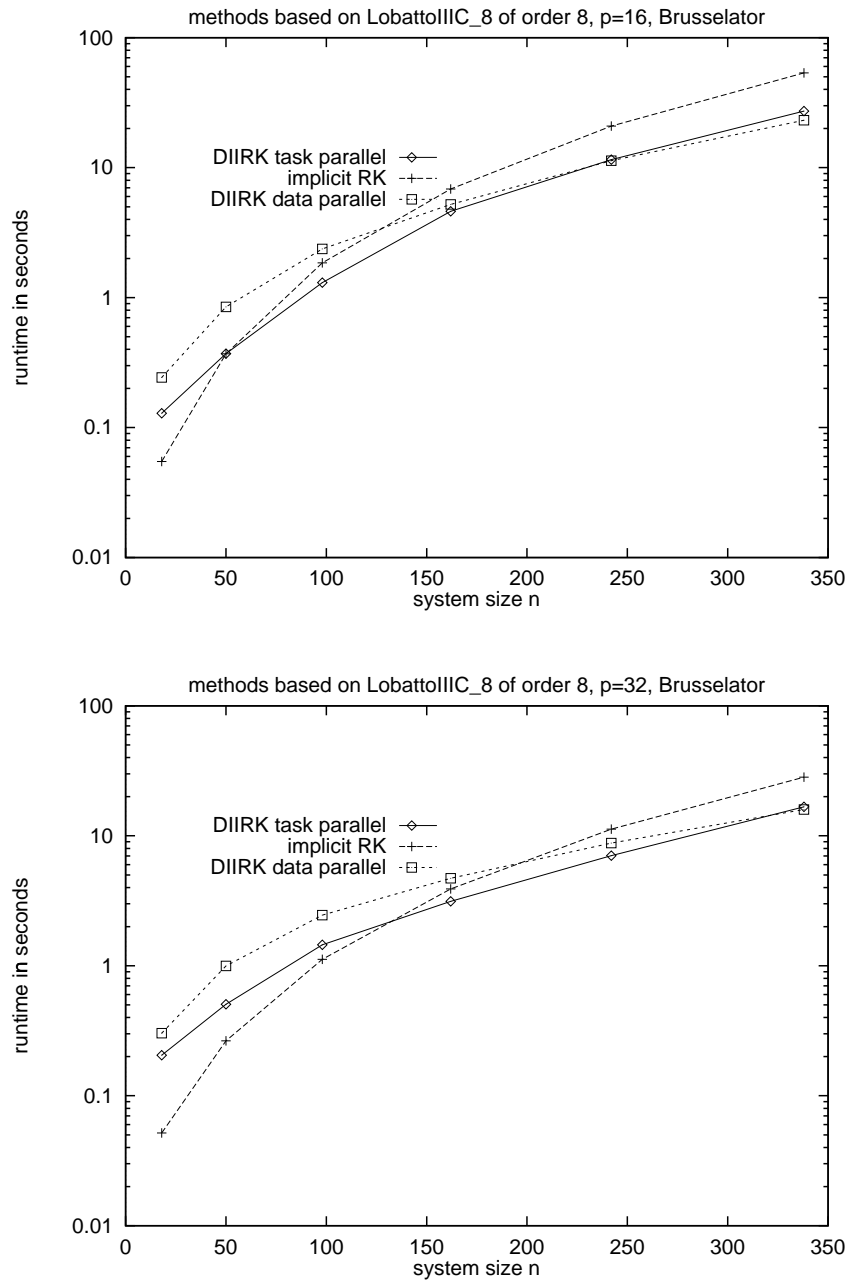


Figure 3.6 Runtimes of methods of order 8 based on Lobatto IIC solving the Brusselator ODE on a T3E using 16 (top) and 32 (bottom) processors.



## 4 PARALLEL SPACE DECOMPOSITION FOR MINIMIZATION OF NONLINEAR FUNCTIONALS

Andreas Frommer,

*University of Wuppertal  
Department of Mathematics  
D-42097 Wuppertal, Germany*

frommer@math.uni-wuppertal.de

Rosemary A. Renaut

*Arizona State University  
Department of Mathematics  
Tempe, AZ 85287-1804*

renaut@asu.edu

**Abstract:** We present an approach for parallel space decomposition which facilitates minimization of sufficiently smooth non-linear functionals with or without constraints on the variables. The framework for the spatial decomposition unites existing approaches from parallel optimization, **parallel variable distribution**, and finite elements, **Schwarz methods**. Additive and multiplicative algorithms based on the spatial decomposition are described. Convergence theorems are also presented, from which convergence for the case of convex functionals, and hence linear least squares problems, follows immediately.

## 4.1 INTRODUCTION

For our research we are concerned with the development of parallel algorithms for the solution of the problem

$$\min_{x \in X} f(x), \quad g(x) \geq 0, \quad (4.1)$$

where  $f$  and  $g$  are smooth nonlinear functionals on  $\mathbf{R}^n$ , and  $X$  is a closed nonempty set in  $\mathbf{R}^n$ . The work presented here provides **additive** and **multiplicative** algorithms for the solution of (4.1). A convergence theory provides for convergence when  $f$  is *convex* and the problem is constrained only by the requirement that  $x \in X$ , where  $X$  is also *convex*.

Approaches for parallel solution of (4.1) typically address the problem in one of two ways. There is a vast literature on appropriate sequential methods for the solution of optimization problems, with the format of the algorithms heavily dependent on the precise formulation and characteristic properties of the defining functionals. Intrinsic to most of these strategies are efficient kernels for iterative linear solvers. Thus, one approach for parallelization is to focus on the most computationally intensive aspects of the algorithm and seek to parallelize these kernels. This has the advantage of maintaining all convergence characteristics of the underlying methods, which have been developed through many years of experience and research. On the other hand, some of the advantages of parallelization may be lost if significant portions of the code are still required to run in a sequential fashion. The alternative is to seek to design new algorithms for optimization which are particularly suited to parallelism and stand to offer high parallel efficiency. In this framework there is potential for the development of completely new strategies which could improve on current understanding of sequential formulations. At the same time, there is also the need to develop a theory to describe the convergence characteristics of any new method. Thus the second approach for parallelism does present a potentially greater improvement in parallel efficiency, but with the disadvantage that the significant progress that has been made in the understanding of sequential optimization may not carry over.

The work presented in this paper seeks to develop an approach for parallelism which is based on spatial decomposition and minimization on subdomains utilizing any method of choice for the subdomain minimizations. A key component of the resulting parallel algorithm also relies on an effective recombination, at minimal sequential cost, of local solutions to generate an iterative update of the global solution. Three methods for the update will be described. This work extends research by [9] and [8], on the use of iterative space decomposition methods for *strongly convex* functionals and provides a unifying framework which connects the approach to the ideas of [3] on the *parallel variable distribution* (PVD) algorithm.

In the next section we describe the spatial decomposition and present the algorithms. Convergence results for the unconstrained case, both for exact and inexact local solutions, are given in Section 4.3. The constrained case is reviewed in Section 4.4. Proofs of all results, further discussion, and some numerical computations are discussed in [4].

## 4.2 PARALLEL SPACE DECOMPOSITION

Throughout the whole paper we denote by  $V_i$ ,  $i = 1, \dots, m$ , a collection of  $m$  (non-trivial) subspaces of  $V = \mathbf{R}^n$  which span the whole of  $V$ , i.e.

$$V = \sum_{i=1}^m V_i. \quad (4.2)$$

We do not assume that this sum is direct, so a vector in  $V$  may have several different representations as a sum of components from the  $V_i$ . The  $V_i$ ,  $i = 1, \dots, m$  are termed a *space decomposition* of  $V$ . We consider all spaces  $V_i$ , as well as  $V$ , as being equipped with the Euclidean norm  $\|\cdot\|$  from  $V$ .

With each of the spaces  $V_i$  we associate the linear and injective embedding operator  $P_i : V_i \rightarrow V$  which maps  $x$  as an element of  $V_i$  on  $x$  as an element of  $V$ , and the corresponding surjective restriction  $R_i = P_i^T : V \rightarrow V_i$ . Properties of these operators are described in [4].

### 4.2.1 Algorithms

For the unconstrained problem (4.1) we formulate  $m$  unconstrained local minimization problems for the auxiliary functions  $f_i : V_i \rightarrow \mathbf{R}$ ,  $i = 1, \dots, m$ ,

$$\min_{y_i^k \in V_i} f_i(y_i) = \min_{y_i^k \in V_i} f(x + P_i^k y_i), \quad i = 1, \dots, m, \quad (4.3)$$

where  $V = \sum_{i=1}^m V_i^k$  is a given sequence of space decompositions with corresponding prolongations  $P_i^k$ . Two iterative algorithms, one *additive* and the other *multiplicative*, for the solution of the unconstrained global minimization problem are based on the solution of these local minimizations. Here, and throughout, the superscript  $k$  on any variable indicates the value at the  $k$ th iteration, while index  $i$  is associated with the corresponding subspace  $V_i^k$ .

**Algorithm 1.** (Additive variant for  $m$  processors)

choose  $x^0 \in V$  and  $\beta_1, \dots, \beta_m > 0$  with  $\sum_i \beta_i = 1$

For  $k = 0, 1, \dots$  (! until  $\nabla f(x^k) = 0$ )

choose a space decomposition  $V = \sum_{i=1}^m V_i^k$

For  $i = 1, \dots, m$

Compute  $y_i^k \in V_i^k$  such that

$$f(x^k + P_i^k y_i^k) = \min_{y_i \in V_i^k} f(x^k + P_i^k y_i) \quad (4)$$

(! local minimization)

$$x^{i,k} = x^k + P_i^k y_i^k$$

Determine  $\alpha_i^k$ ,  $i = 1, \dots, m$ , form update

(! synchronization)

$$x^{k+1} = x^k + \sum_{i=1}^m \alpha_i^k P_i^k y_i^k \quad (5)$$

such that

$$f(x^{k+1}) \leq \sum_{i=1}^m \beta_i f(x^{i,k}) \quad (6)$$

End

End.

We consider three approaches for fulfilling (4.2.1) at each global update.

i) The *optimal* strategy . Determine  $\alpha_i^k, \dots, \alpha_m^k$  by solving the  $m$ -dimensional minimization problem

$$\min_{(\alpha_1, \dots, \alpha_m) \in \mathbf{R}^m} f(x^k + \sum_i \alpha_i P_i^k y_i^k). \quad (4.7)$$

ii) The *selection* strategy . Determine  $t$  such that

$$f(x^{t,k}) = \min_{i=1}^m f(x^{i,k}).$$

Then, with  $\alpha_t^k = 1$  and  $\alpha_i^k = 0, i \neq t$ ,

$$x^{k+1} = x^{t,k}. \quad (4.8)$$

iii) The *convex combination* strategy. For convex  $f$  form the convex update

$$x^{k+1} = \sum_{i=1}^m \beta_i x^{i,k} = x^k + \sum_{i=1}^m \beta_i P_i^k y_i^k, \quad (4.9)$$

with  $\alpha_i^k = \beta_i, i = 1, \dots, m$ .

Note that this algorithm may also be seen as a coordinate descent method [1] if the choice  $\alpha_i^k = 1$  is made for all choices of  $i$  and  $k$ . We do not consider this strategy for global update because the descent condition (4.2.1) need not be satisfied and hence convergence is not guaranteed. In addition, when (4.9) is used, the algorithm relates to the methods of [7], and [8], except that here the space decomposition is not required to be direct. When  $f$  is restricted to be a quadratic functional the above algorithm may also be interpreted as the classical additive Schwarz iterative solver for the equivalent linear system, e.g. [5].

A multiplicative version of the algorithm, which does not facilitate complete parallelism unless subproblem independence provides a suitable coloring, is given as follows:

**Algorithm 2.** (Multiplicative variant)

Choose  $x^0 \in \mathbf{R}^n$ ,

For  $k = 0, 1, \dots$ , (! until  $\nabla f(x^k) = 0$ )

Choose a space decomposition  $V = \sum_i V_i^k$ .



```

For  $i = 1, \dots, m$ 
  Compute  $y_i^k \in V_i^k$  such that
     $f\left(x^k + \omega \sum_{j=1}^{i-1} P_j^k y_j^k + P_i^k y_i^k\right)$ 
     $= \min_{y_i \in V_i} f\left(x^k + \omega \sum_{j=1}^{i-1} P_j^k y_j^k + P_i^k y_i\right)$  (10)
End
 $x^{k+1} = x^k + \omega \sum_{j=1}^m P_j^k y_j^k$ 
End

```

Here  $\omega \in \mathbf{R}$  is an *a priori* relaxation factor. The choice  $\omega = 1$  corresponds to a Gauss-Seidel variant, whereas  $\omega \neq 1$  gives the SOR variant of the algorithm.

### 4.3 CONVERGENCE THEORY

#### 4.3.1 The additive algorithm

We will now give convergence results for the additive algorithm and certain variants in the case that the functional  $f$  has a Lipschitz-continuous gradient, i.e. there exists  $K > 0$  such that

$$\|\nabla f(y) - \nabla f(x)\| \leq K \|y - x\|, \quad \forall x, y \in \mathbf{R}^n. \quad (4.11)$$

We write this as  $f \in LC_K^1(V)$ . Note that we view the gradient primarily as a linear mapping from  $V$  to  $\mathbf{R}^n$ , i.e. as a ‘row vector’. Whenever we need to identify  $\nabla f(x)$  with its dual from  $V$  we write  $\nabla f(x)^T$ .

We start with the following general result.

**Theorem 1 (Convergence of the additive algorithm)** *Let  $f \in LC_K^1(V)$  be bounded from below and let  $V = \sum_{i=1}^m V_i^k$  be a sequence of space decompositions such that*

$$\sum_{i=1}^m \|R_i^k x\|^2 \geq c \cdot \|x\|^2 \quad \text{for all } x \in V \text{ and } k = 0, 1, \dots \quad (4.12)$$

*for some  $c > 0$ . Then every accumulation point of the sequence  $\{x^k\}$  generated by Algorithm 1 is stationary and  $\lim_{k \rightarrow \infty} \nabla f(x^k) = 0$ .*

A result on the rate of convergence for strongly convex  $f$  is then immediate from the definition of strong convexity and a result on linear convergence given in [3].

**Corollary 1** *Assume that, in addition to the hypothesis of Theorem 1, the functional  $f$  is strongly convex with constant  $C$ . Then the sequence of iterates  $\{x^k\}$  converges to  $x^*$ , the unique minimizer of  $f$ , at the linear root rate*

$$\|x^k - x^*\| \leq \left( \frac{2}{C} (f(x^0) - f(x^k)) \right)^{1/2} \left( 1 - \frac{c\beta C^2}{K^2} \right)^{k/2}.$$

The PVD framework of [3] assumes a forget-me-not term in the local minimization by restricting each  $V_i^k$  to the form

$$V_i^k = W_i + W^k, \quad (4.13)$$

where the  $W_i$  form a non-overlapping orthogonal space decomposition of  $V$  with each  $W_i$  independent of  $k$  and spanned by Cartesian unit vectors. The spaces  $W^k$  are of dimension  $m$  and are spanned by vectors, one from each  $W_i$ . The following corollary then applies, and improves on the results given in [2], [3] and [6] by showing that the restrictions on the forget-me-not terms can be dispensed with.

**Corollary 2** *Let  $f \in LC_K^1(V)$  be bounded from below. Then every accumulation point of the sequence  $\{x^k\}$  generated by the PVD algorithm is stationary and  $\lim_{k \rightarrow \infty} \nabla f(x^k) = 0$ . Moreover, if  $f$  is strongly convex,  $\lim_{k \rightarrow \infty} x^k = x^*$ , where  $x^*$  is the unique minimizer of  $f$ .*

### 4.3.2 Inexact local solutions

The results can also be relaxed to permit acceptance of inexact local solutions in (4.2.1) by observing that for the proof of Theorem 1 we need the local solutions to satisfy

$$f(x^k) - f(x^{i,k}) \geq \frac{1}{2K} \|R_i \nabla f(x^k)^T\|^2,$$

which is a local sufficient descent condition.

**Theorem 2 (Convergence of the inexact additive algorithm)** *Let  $f \in LC_K^1(V)$  be bounded from below. Let  $V = \sum_{i=1}^m V_i^k$  be a sequence of space decompositions such that*

$$\sum_{i=1}^m \|R_i^k x\|^2 \geq c \cdot \|x\|^2 \text{ for all } x \in V \text{ and } k = 0, 1, \dots,$$

*for some  $c > 0$ . Let  $\alpha > 0$  and assume that in Algorithm 4.2.1, instead of step (4.2.1) we accept ‘inexact’ solutions  $x^{i,k}$  to the local minimization problem whenever*

$$f(x^k) - f(x^{i,k}) \geq \alpha \cdot \|R_i^k \nabla f(x^k)^T\|^2. \quad (4.14)$$

*Then every accumulation point of the sequence  $\{x^k\}$  generated by the modified algorithm is stationary and  $\lim_{k \rightarrow \infty} \nabla f(x^k) = 0$ . Moreover, if  $f$  is strongly convex and*

its gradient is Lipschitz-continuous, then  $\lim_{k \rightarrow \infty} x^k = x^*$ , the unique minimizer of  $f$ .

This result then also provides the convergence result with *under* relaxation of *exact* local solutions for strongly convex functionals (see [7]), and with *over* relaxation for quadratic functionals.

**Corollary 3** Assume that  $f \in LC_K^1(V)$  is strongly convex and let the space decompositions  $V = \sum_{i=1}^m V_i^k$  be as in Theorem 2.

- (i) Let  $\gamma_1, \dots, \gamma_m$  be positive numbers such that  $\gamma := \sum_{i=1}^m \gamma_i \leq 1$  and assume that for all  $k$  the synchronization step (4.2.1) in Algorithm 4.2.1 is replaced by

$$x^{k+1} = x^k + \sum_{i=1}^m \gamma_i P_i^k y_i^k. \quad (4.15)$$

Then  $\lim_{k \rightarrow \infty} x^k = x^*$ , the unique minimizer of  $f$ .

- (ii) In the special case when  $f$  is a quadratic functional part (i) holds with  $\gamma \leq 1$  replaced by  $\gamma < 2$ .

### 4.3.3 The multiplicative algorithm

The following convergence results, largely equivalent to those for the additive variant, can also be obtained, but with in each case the assumption that  $f$  is strongly convex.

**Theorem 3 (Convergence of the multiplicative algorithm)** Let  $f \in LC_K^1(V)$  be strongly convex. Assume that  $V = \sum_{i=1}^m V_i^k$  is a sequence of space decompositions such that

$$\sum_{i=1}^m \|R_i^k x\|^2 \geq c \cdot \|x\|^2 \text{ for all } x \in V \text{ and } k = 0, 1, \dots,$$

for some  $c > 0$ . Then the sequence  $\{x^k\}$  of iterates produced by Algorithm 4.6 with  $\omega = 1$  converges to  $x^*$ , the unique minimizer of  $f$  in  $V$ .

The above theorem has been given in [7] for the case of minimization subject to block separable constraints on a closed convex set in  $V$ , and for which the space decomposition is independent of  $k$ . To illustrate just one new result covered by our general theorem, let us note that, due to (4.13), we get convergence for the multiplicative variant of the PVD method.

**Theorem 4 (Convergence of inexact multiplicative algorithm)** Let  $f \in LC_K^1(V)$  be strongly convex. Let  $V = \sum_{i=1}^m V_i^k$  be a sequence of space decompositions such that

$$\sum_{i=1}^m \|R_i^k x\|^2 \geq c \cdot \|x\|^2 \text{ for all } x \in V \text{ and } k = 0, 1, \dots,$$

for some  $c > 0$ . Let  $\omega = 1$  and assume that in Algorithm 4.6, instead of step (4.6) we accept 'inexact' solutions  $y_i^k$  to the minimization problem whenever

$$f(x^{i-1,k}) - f(x^{i,k}) \geq \alpha \cdot \|R_i^k \nabla f(x^{i-1,k})^T\|^2, \quad (4.16)$$

$$i = 1, \dots, m,$$

where  $x^{i,k} = x^k + \sum_{j=1}^i P_j^k y_j^k$ ,  $i = 0, \dots, m$  and  $\alpha > 0$  is fixed. Then  $\lim_{k \rightarrow \infty} x^k = x^*$ , the unique minimizer of  $f$  in  $V$ .

#### 4.4 THE CONSTRAINED PVD ALGORITHM

The PVD theory presented in [3] provides convergence of Algorithm 1 with synchronization steps (4.7) or (4.8) for the case of the block separable constrained problem, i.e., for which  $X$  in (4.1) is assumed to be a Cartesian product of closed convex sets, [3, Theorem 3.7]. In order to solve the constrained problem it is necessary to introduce modifications to the algorithms that incorporate the appropriate constraint conditions. For now we consider the general convex-constrained case, where  $X$  is a closed convex set, from which the result for the block separable case can be deduced. Then in Algorithm 1 the local minimizations (4.2.1) are replaced by

$$\begin{aligned} \min_{y_i \in V_i} f_i(y_i) &= \min_{y_i \in V_i} f(x^k + P_i y_i) \\ x^k + P_i y_i &\in X \quad x^k + P_i y_i \in X. \end{aligned} \quad (4.17)$$

Convergence for the constrained problem is defined in terms of the convergence to a stationary point  $\bar{x}$  at which the minimum principle necessary optimality condition

$$\bar{x} \in X \quad \text{and} \quad \nabla f(\bar{x})(y - \bar{x}) \geq 0 \quad \forall y \in X \quad (4.18)$$

is satisfied. Equivalently,

$$r(\bar{x}) = 0, \quad (4.19)$$

where  $r$  is the projected gradient residual function defined by

$$r(x) := x - [x - \nabla f(x)^T]_+, \quad (4.20)$$

and  $[x]_+$  represents the orthogonal projection map for element  $x \in V$  onto the set  $X$ .

We thus obtain convergence for general convex  $X$ .

**Theorem 5 (Convergence for general convex  $X$ )** *Let  $f \in LC_K^1(V)$  be strongly convex. Then the sequence of iterates  $\{x^k\}$  produced by Algorithm 1 with local minimizations (4.17) converges to  $x^*$ , the unique minimizer of  $f(x)$  over  $X \subset V$ .*

Extensions for inexact local solutions, and to give rate of convergence results, may also be derived as in the unconstrained case.

### 4.4.1 Acknowledgments

The research of the second author was supported under grant NSF DMS9402943.

### References

- [1] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [2] J.E. Dennis and T. Steihaug, *A Ferris-Mangasarian Technique Applied to Linear Least Squares Problems*, CRPC-TR 98740, Center for Research on Parallel Computation, Rice University, 1998.
- [3] M.C. Ferris and O.L. Mangasarian, *Parallel Variable Distribution*, SIAM J. Optim. **4** (1994), 815–832.
- [4] A. Frommer and R. A. Renaut, *A Unified Approach to Parallel Space Decomposition Methods*, 1999.
- [5] W. Hackbusch, *Iterative Methods for Large Sparse Linear Systems*, Springer, Heidelberg, 1993.
- [6] M.V. Solodov, *New Inexact Parallel Variable Distribution Algorithms*, Comp. Optim. and Appl. **7** (1997), 165–182.
- [7] X.C. Tai, *Parallel function and space decomposition methods—Part II, Space decomposition*, Beijing Mathematics, 1, Part 2, (1995), 135–152.
- [8] X.C. Tai and M. Espedal, *Rate of Convergence of Some Space Decomposition Methods for Linear and Nonlinear Problems*, SIAM J. Numer. Anal. **35** (1998), 1558–1570 .
- [9] J. Xu, *Iterative Methods by Space Decomposition and Subspace Correction*, SIAM Review **34** (1992), 581–613.



## 5 REDUCING GLOBAL SYNCHRONIZATION IN THE BICONJUGATE GRADIENT METHOD

H. Martin Buecker,\*

*Institute for Scientific Computing  
Aachen University of Technology  
D-52056 Aachen, Germany*

buecker@sc.rwth-aachen.de

Manfred Sauren

*MicroStrategy Deutschland GmbH  
Kölner Straße 263  
D-51149 Köln, Germany*

msauren@strategy.com

**Abstract:** Starting from a specific implementation of the Lanczos biorthogonalization algorithm, an iterative process for the solution of systems of linear equations with general non-Hermitian coefficient matrix is derived. Due to the orthogonalization of the underlying Lanczos process the resulting iterative scheme involves inner products leading to global communication and synchronization on parallel processors. For massively parallel computers, these effects cause considerable delays often preventing the scalability of the implementation. In the process proposed, all inner product-like operations of an

---

\*This work was produced while both authors were in residence at the Central Institute for Applied Mathematics, Forschungszentrum Jülich GmbH, D-52425 Jülich, Germany.

iteration step are independent such that the implementation consists of only a single global synchronization point per iteration. In exact arithmetic, the process is shown to be mathematically equivalent to the biconjugate gradient method. The efficiency of this new variant is demonstrated by numerical experiments on a PARAGON system using up to 121 processors.

## 5.1 INTRODUCTION

Storage requirements as well as computation times often set limits to direct methods for the solution of systems of linear equations arising from large scale applications in science and engineering. If the coefficient matrix is sparse, which typically is the case for systems resulting from discretization of partial differential equations by finite differences or finite elements, iterative methods offer an alternative. A popular class of iterative methods is the family of Krylov subspace methods [8, 18] involving the coefficient matrix only in the form of matrix-by-vector products. These methods basically consist of the generation of a suitable basis of a vector space called Krylov subspace and the choice of the actual iterate within that space.

In this note, a Krylov subspace method for non-Hermitian linear systems is derived that generates the basis of its primary subspace by means of the Lanczos biorthogonalization algorithm [15]. Due to the orthogonalization of this underlying procedure the resulting process contains inner products that lead to global communication on parallel processors, i.e., communication of all processors at the same time. On the contrary, for large sparse matrices, parallel matrix-by-vector products can be implemented with communication between only nearby processors. This is the reason why scalability of an implementation on massively parallel processors is usually prevented entirely by the computation of inner product-like operations. There are two strategies to remedy the performance degradation which, of course, can be combined. The first is to restructure the code such that most communication is overlapped with useful computation. The second is to eliminate data dependencies such that several inner products are computed simultaneously. This note is concerned with the latter strategy in order to reduce the number of global synchronization points. A *global synchronization point* is defined as the locus of an algorithm at which all local information has to be globally available in order to continue the computation.

The organization of this note which is an extended version of [3] is as follows. Section 5.2 is concerned with a sketch of a specific version of the Lanczos biorthogonalization algorithm suitable for massively parallel computing. In Sect. 5.3, this version is used to derive a new process for the solution of systems of linear equations that is shown to be mathematically equivalent to the biconjugate gradient method (BCG) [16, 6]. A rescaled version of the original BCG is reviewed in Sect. 5.4 and used as a contrast to report numerical experiments and timings on a PARAGON system using up to 121 processors in Sect. 5.5.



## 5.2 UNSYMMETRIC Lanczos ALGORITHM

The Lanczos biorthogonalization algorithm [15] indexLanczos biorthogonalization algorithm often called unsymmetric Lanczos algorithm (Lanczos algorithm hereafter) was originally proposed to reduce a matrix to tridiagonal form. We here focus on a different application, namely the computation of two bases of two Krylov subspaces. Given a general non-Hermitian matrix  $\mathbf{A} \in \mathbb{C}^{N \times N}$  and two starting vectors  $\mathbf{v}_1, \mathbf{w}_1 \in \mathbb{C}^N$  satisfying  $\mathbf{w}_1^T \mathbf{v}_1 = 1$ , the Lanczos algorithm generates two finite sequences of vectors  $\{\mathbf{v}_n\}_{n=1,2,3,\dots}$  and  $\{\mathbf{w}_n\}_{n=1,2,3,\dots}$  with the following three properties:

$$\mathbf{v}_n \in \mathcal{K}_n(\mathbf{v}_1, \mathbf{A}) , \quad (5.1)$$

$$\mathbf{w}_n \in \mathcal{K}_n(\mathbf{w}_1, \mathbf{A}^T) , \quad (5.2)$$

$$\mathbf{w}_m^T \mathbf{v}_n = \begin{cases} 0 & \text{if } n \neq m , \\ 1 & \text{if } n = m , \end{cases} \quad (5.3)$$

where  $\mathcal{K}_n(\mathbf{y}, \mathbf{A}) = \text{span}\{\mathbf{y}, \mathbf{A}\mathbf{y}, \dots, \mathbf{A}^{n-1}\mathbf{y}\}$  denotes the  $n$ th Krylov subspace generated by the matrix  $\mathbf{A}$  and the vector  $\mathbf{y}$ . The vectors  $\mathbf{v}_n$  and  $\mathbf{w}_n$  are called Lanczos vectors and their relation (5.3) is commonly referred to as biorthogonality. If the Lanczos vectors are put as columns into matrices

$$\mathbf{V}_n = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n] \quad \text{and} \quad \mathbf{W}_n = [\mathbf{w}_1 \ \mathbf{w}_2 \ \cdots \ \mathbf{w}_n]$$

the Lanczos algorithm is characterized—up to scaling—by the three equations

$$\mathbf{W}_n^T \mathbf{V}_n = \mathbf{I}_n , \quad (5.4a)$$

$$\mathbf{A} \mathbf{V}_n = \mathbf{V}_n \mathbf{T}_n + \gamma_{n+1} \mathbf{v}_{n+1} \mathbf{e}_n^T , \quad (5.4b)$$

$$\mathbf{A}^T \mathbf{W}_n = \mathbf{W}_n \mathbf{T}_n^T + \beta_{n+1} \mathbf{w}_{n+1} \mathbf{e}_n^T , \quad (5.4c)$$

where in addition to the  $n \times n$  identity matrix  $\mathbf{I}_n$  and its last row  $\mathbf{e}_n^T$  the  $n \times n$  tridiagonal matrix

$$\mathbf{T}_n = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \gamma_2 & \alpha_2 & \beta_3 & & \\ & \gamma_3 & & \ddots & \\ & & & \ddots & \ddots & \\ & & & & \ddots & \beta_n \\ & & & & \gamma_n & \alpha_n \end{pmatrix}$$

is used. The tridiagonal structure of  $\mathbf{T}_n$  leads to three-term recurrences for the generation of the Lanczos vectors which is immediately apparent if the algorithm is formulated in vector notation rather than in matrix notation; see [11] for details. We further state that the formulation (5.4) involves two global synchronization points per iteration which readily can be reduced to a single one by a simple reorganization of the statements [14]. We stress that this version opens the alternative either to scale the  $\mathbf{v}_n$ 's or  $\mathbf{w}_n$ 's but not both.

It has been pointed out by several authors [19, 17, 8, 10] that in practice one should have the possibility to scale both sequences of Lanczos vectors appropriately, e.g.,

$$\|\mathbf{v}_n\|_2 = \|\mathbf{w}_n\|_2 = 1, \quad n = 1, 2, 3, \dots, \quad (5.5)$$

in order to avoid over- or underflow.

Rather than performing the necessary changes to the above three-term procedure which can be found in [9], we here concentrate on a different variant involving coupled two-term recurrences. This formulation that is based on the LU decomposition of the tridiagonal matrix  $\mathbf{T}_n$  is, in exact arithmetic, mathematically equivalent to the three-term procedure. In finite precision arithmetic, numerical experiments [10] suggest to favor the coupled two-term procedure when used as the underlying process of an iterative method for the solution of linear systems. Therefore, we start our discussion with a summary of the findings given in [2] where a specific variant of the Lanczos algorithm based on coupled two-term recurrences is derived. This variant offers the possibility to scale both sequences of Lanczos vectors and it involves only a single global synchronization point per iteration. We remark that each iteration of the corresponding version with unit length scaling of both Lanczos vectors given in [10] consists of three global synchronization points which can be reduced to two without considerable difficulties—but not to a single one.

We here state the matrix equations that characterize the variant of the Lanczos algorithm developed in [2] and give some comments afterwards:

$$\mathbf{W}_n^T \mathbf{V}_n = \mathbf{D}_n, \quad (5.6a)$$

$$\mathbf{V}_n = \mathbf{P}_n \mathbf{U}_n, \quad (5.6b)$$

$$\mathbf{A} \mathbf{P}_n = \mathbf{V}_{n+1} \mathbf{L}_n, \quad (5.6c)$$

$$\tilde{\mathbf{Q}}_n = \mathbf{W}_n \mathbf{D}_n^{-1} \mathbf{U}_n^T + \frac{\mu_{n+1}}{\delta_{n+1}} \mathbf{w}_{n+1} \mathbf{e}_n^T, \quad (5.6d)$$

$$\mathbf{A}^T \mathbf{W}_n = \tilde{\mathbf{Q}}_{n-1} \mathbf{L}_{n-1}^T \mathbf{D}_n + \tau_n \delta_n \tilde{\mathbf{q}}_n \mathbf{e}_n^T. \quad (5.6e)$$

The possibility to scale both sequences of Lanczos vectors requires to replace the biorthogonality relation (5.4a) by (5.6a) where the diagonal matrix

$$\mathbf{D}_n = \text{diag}(\delta_1, \delta_2, \dots, \delta_n)$$

with  $\delta_i \neq 0$  for  $i = 1, 2, \dots, n$  is used. The process is said to be based on coupled two-term recurrences because of the bidiagonal structure of the matrices

$$\mathbf{L}_n = \begin{pmatrix} \tau_1 & & & \\ \gamma_2 & \ddots & & \\ & \ddots & \tau_n & \\ & & \gamma_{n+1} & \end{pmatrix} \in \mathbb{C}^{(n+1) \times n} \quad (5.7)$$

and

$$\mathbf{U}_n = \begin{pmatrix} 1 & \mu_2 & & \\ & 1 & \ddots & \\ & & \ddots & \mu_n \\ & & & 1 \end{pmatrix} \in \mathbb{C}^{n \times n} .$$

We remark that these matrices are the factors of the LU decomposition of the tridiagonal matrix  $\mathbf{T}_n$  of the three-term procedure if the last row of  $\mathbf{L}_n$  is eliminated. In addition to the Lanczos vectors, two more vector sequences,  $\{\mathbf{p}_n\}_{n=1,2,3,\dots}$  and  $\{\tilde{\mathbf{q}}_n\}_{n=1,2,3,\dots}$ , are generated giving rise to matrices

$$\mathbf{P}_n = [\mathbf{p}_1 \ \mathbf{p}_2 \ \cdots \ \mathbf{p}_n] \quad \text{and} \quad \tilde{\mathbf{Q}}_n = [\tilde{\mathbf{q}}_1 \ \tilde{\mathbf{q}}_2 \ \cdots \ \tilde{\mathbf{q}}_n] .$$

Equations (5.6b) and (5.6c) show that the generation of the vector sequences  $\mathbf{v}_n$  and  $\mathbf{p}_n$  is coupled and so are  $\mathbf{w}_n$  and  $\tilde{\mathbf{q}}_n$ . The complete derivation of this variant of the Lanczos algorithm is given in [2]. We finally stress that there are look-ahead techniques [19, 17, 12, 9, 13] preventing the Lanczos algorithm from breaking down. Although any implementation will benefit from such techniques they are beyond the scope of this note.

### 5.3 A PARALLEL VARIANT OF BCG

The Lanczos algorithm is now used as the underlying process of an iterative method for the solution of systems of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} , \tag{5.8}$$

where  $\mathbf{A} \in \mathbb{C}^{N \times N}$  and  $\mathbf{x}, \mathbf{b} \in \mathbb{C}^N$ . One of the features of the Lanczos algorithm is the fact that the Lanczos vectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$  form a basis of the Krylov subspace generated by  $\mathbf{A}$  and the starting vector  $\mathbf{v}_1$ . Since Krylov subspace methods for the solution of (5.8) are characterized by

$$\mathbf{x}_n \in \mathbf{x}_0 + \mathcal{K}_n(\mathbf{r}_0, \mathbf{A}) ,$$

where  $\mathbf{x}_0$  is any initial guess and  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$  is the corresponding residual vector, relation (5.1) raises hopes to derive a Krylov subspace method if the Lanczos algorithm is started with

$$\mathbf{v}_1 = \frac{1}{\gamma_1} \mathbf{r}_0 , \tag{5.9}$$

where  $\gamma_1 = \|\mathbf{r}_0\|$  is a scaling factor. Then, the  $n$ th iterate is of the form

$$\mathbf{x}_n = \mathbf{x}_0 + \mathbf{V}_n \mathbf{z}_n , \tag{5.10}$$

where as before the columns of  $\mathbf{V}_n \in \mathbb{C}^{N \times n}$  are nothing but the Lanczos vectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$  and  $\mathbf{z}_n \in \mathbb{C}^n$  is a parameter vector to be fixed later. The goal of any Krylov subspace method is in some sense to drive the residual vector

$$\mathbf{r}_n = \mathbf{b} - \mathbf{A}\mathbf{x}_n$$

to the zero vector. Using (5.6b) and introducing

$$\mathbf{y}_n = \mathbf{U}_n \mathbf{z}_n, \quad (5.11)$$

the iterates according to (5.10) are reformulated in terms of  $\mathbf{y}_n$  instead of  $\mathbf{z}_n$  giving

$$\mathbf{x}_n = \mathbf{x}_0 + \mathbf{P}_n \mathbf{y}_n. \quad (5.12)$$

On account of (5.9) and (5.6c) the corresponding residual vector is

$$\mathbf{r}_n = \mathbf{V}_{n+1} \left( \gamma_1 \mathbf{e}_1^{(n+1)} - \mathbf{L}_n \mathbf{y}_n \right), \quad (5.13)$$

where  $\mathbf{e}_1^{(n+1)} = (1, 0, \dots, 0)^T \in \mathbb{C}^{n+1}$ . Generating the Krylov subspace by means of the Lanczos algorithm and fixing  $\mathbf{y}_n$ , and so implicitly  $\mathbf{z}_n$  by (5.11), an iterative method can be derived. Let  $\hat{y}_i$  denote the  $i$ th component of  $\mathbf{y}_n$ , i.e.,  $\mathbf{y}_n = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n)^T$ . The idea is to drive  $\mathbf{r}_n$  to the zero vector by

$$\hat{y}_i = \begin{cases} \frac{\gamma_1}{\tau_1} & \text{if } i = 1, \\ -\frac{\gamma_i}{\tau_i} \hat{y}_{i-1} & \text{if } i = 2, 3, \dots, n. \end{cases}$$

This choice of  $\mathbf{y}_n$  is motivated by the structure of  $\mathbf{L}_n$  given in (5.7) and zeros out the first  $n$  components of the vector  $\gamma_1 \mathbf{e}_1^{(n+1)} - \mathbf{L}_n \mathbf{y}_n$  in (5.13). The residual vector is then given by

$$\mathbf{r}_n = \mathbf{V}_{n+1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ -\gamma_{n+1} \hat{y}_n \end{pmatrix}. \quad (5.14)$$

It is important to note that the process of fixing  $\mathbf{y}_n$  is easily updated in each iteration step because  $\mathbf{y}_{n-1}$  coincides with the first  $n-1$  components of  $\mathbf{y}_n$ . The corresponding recursion is

$$\mathbf{y}_n = \begin{pmatrix} \mathbf{y}_{n-1} \\ 0 \end{pmatrix} + \kappa_n \mathbf{e}_n, \quad (5.15)$$

where  $\mathbf{e}_n = (0, \dots, 0, 1)^T \in \mathbb{C}^n$  and

$$\kappa_n = -\frac{\gamma_n}{\tau_n} \kappa_{n-1} \quad (5.16)$$

with  $\kappa_0 = -1$ . Inserting (5.15) into (5.12) yields

$$\begin{aligned}\mathbf{x}_n &= \mathbf{x}_0 + \mathbf{P}_{n-1}\mathbf{y}_{n-1} + \kappa_n \mathbf{p}_n \\ &= \mathbf{x}_{n-1} + \kappa_n \mathbf{p}_n .\end{aligned}\tag{5.17}$$

Note that  $\|\mathbf{r}_n\|_2$  is immediately available because, at every iteration step, the last component of  $\mathbf{y}_n$  is  $\kappa_n$  by (5.15). Hence, from (5.14) the relation

$$\|\mathbf{r}_n\|_2 = \|\mathbf{v}_{n+1}\|_2 \cdot |\gamma_{n+1}\kappa_n| \tag{5.18}$$

follows.

Properly putting (5.16) and (5.17) on top of the Lanczos algorithm characterized by equations (5.6) results in Alg. 1. We remark that the underlying Lanczos procedure operates with unit scaling of both sequences of Lanczos vectors, see (5.5), and is explicitly presented in [2, Alg. 3]. In this case, (5.18) simplifies to

$$\|\mathbf{r}_n\|_2 = |\gamma_{n+1}\kappa_n| \tag{5.19}$$

that can be used as a simple stopping criterion. In Alg. 1, the operations leading to global communication on parallel processors are marked with a preceding bullet. The absence of any data dependencies of these operations can be exploited to compute them simultaneously, i.e., by reducing their local partial sums in a single global communication operation.

Having derived the above iterative process for the solution of linear systems, we afterwards realized that Alg. 1 is just a new variant of the biconjugate gradient method (BCG) [16, 6] whose iterates are defined by the Galerkin type condition

$$\mathbf{w}^T \mathbf{r}_n = 0 \quad \text{for all } \mathbf{w} \in \mathcal{K}_n(\mathbf{w}_1, \mathbf{A}^T) , \tag{5.20}$$

where  $\mathbf{w}_1$  is arbitrary, provided  $\mathbf{w}_1^T \mathbf{v}_1 \neq 0$ , but one usually set  $\mathbf{w}_1 = \mathbf{r}_0 / \|\mathbf{r}_0\|_2$  as Alg. 1 implicitly does [2]. To see why (5.20) holds, recall that by (5.2) every vector  $\mathbf{w} \in \mathcal{K}_n(\mathbf{w}_1, \mathbf{A}^T)$  is of the form

$$\mathbf{w} = \mathbf{W}_n \mathbf{s} \quad \text{for some } \mathbf{s} \in \mathbb{C}^n ,$$

where  $\mathbf{W}_n$  is generated by the Lanczos algorithm. Using (5.14) and the above argumentation of  $\kappa_n$  as the last component of  $\mathbf{y}_n$ , the condition (5.20) is equivalent to

$$-\gamma_{n+1}\kappa_n \mathbf{s}^T \mathbf{W}_n^T \mathbf{v}_{n+1} = 0$$

that is satisfied because of the biorthogonality (5.6a).

---

Input  $\mathbf{A}$ ,  $\mathbf{b}$  and  $\mathbf{x}_0$   
 $\mathbf{p}_0 = \mathbf{q}_0 = \mathbf{0}$ ,  $\tilde{\mathbf{v}}_1 = \tilde{\mathbf{w}}_1 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$   
 $\gamma_0 = \xi_0 = 0$ ,  $\tau_0 = \rho_0 \neq 0$ ,  $\kappa_0 = -1$   
 $\gamma_1 = \|\tilde{\mathbf{v}}_1\|_2$ ,  $\xi_1 = \|\tilde{\mathbf{w}}_1\|_2$ ,  $\varrho_1 = \tilde{\mathbf{w}}_1^T \tilde{\mathbf{v}}_1$ ,  $\varepsilon_1 = (\mathbf{A}^T \tilde{\mathbf{w}}_1)^T \tilde{\mathbf{v}}_1$   
**for**  $n = 1, 2, 3, \dots$  **do**  

$$\mu_n = \frac{\gamma_{n-1} \xi_{n-1} \varrho_n}{\gamma_n \tau_{n-1} \varrho_{n-1}}$$

$$\tau_n = \frac{\varepsilon_n}{\varrho_n} - \gamma_n \mu_n$$

$$\mathbf{p}_n = \frac{1}{\gamma_n} \tilde{\mathbf{v}}_n - \mu_n \mathbf{p}_{n-1}$$

$$\mathbf{q}_n = \frac{1}{\xi_n} \mathbf{A}^T \tilde{\mathbf{w}}_n - \frac{\gamma_n \mu_n}{\xi_n} \mathbf{q}_{n-1}$$

$$\tilde{\mathbf{v}}_{n+1} = \mathbf{A} \mathbf{p}_n - \frac{\tau_n}{\gamma_n} \tilde{\mathbf{v}}_n$$

$$\tilde{\mathbf{w}}_{n+1} = \mathbf{q}_n - \frac{\tau_n}{\xi_n} \tilde{\mathbf{w}}_n$$

- $\gamma_{n+1} = \|\tilde{\mathbf{v}}_{n+1}\|_2$
- $\xi_{n+1} = \|\tilde{\mathbf{w}}_{n+1}\|_2$
- $\varrho_{n+1} = \tilde{\mathbf{w}}_{n+1}^T \tilde{\mathbf{v}}_{n+1}$
- $\varepsilon_{n+1} = (\mathbf{A}^T \tilde{\mathbf{w}}_{n+1})^T \tilde{\mathbf{v}}_{n+1}$

$$\kappa_n = -\frac{\gamma_n}{\tau_n} \kappa_{n-1}$$

$$\mathbf{x}_n = \mathbf{x}_{n-1} + \kappa_n \mathbf{p}_n$$
**if**  $(|\gamma_{n+1} \kappa_n| < \text{tolerance})$  **then STOP**  
**endfor**

---

**Algorithm 1:** A parallel variant of the biconjugate gradient method

Although the history of the biconjugate gradient method dates back to the 1950s, we are unaware of any implementation of this method with the properties of the above procedure: The possibility to scale both sequences of Lanczos vectors and the independence of all inner product-like operations leading to only a single global synchronization point per iteration on parallel processors. The mathematical equivalence of the principal idea given here and the biconjugate gradient method is mentioned in [7, 4] although applied to the Lanczos algorithm based on three-term recurrences. We finally remark that, in a slightly modified form, the procedure introduced here turns out to be a special case of a new iterative method minimizing a factor of  $\mathbf{r}_n$  with respect to the  $p$ -norm that will be presented elsewhere. Moreover, the Lanczos

algorithm characterized by equations (5.6) is also useful to derive a parallel variant of another iterative method [1].

---

```

Input  $\mathbf{A}$ ,  $\mathbf{b}$  and  $\mathbf{x}_0$ 
 $\mathbf{p}_0 = \mathbf{q}_0 = \mathbf{0}$ ,  $\mathbf{v}_1 = \mathbf{w}_1 = (\mathbf{b} - \mathbf{A}\mathbf{x}_0)/\|\mathbf{b} - \mathbf{A}\mathbf{x}_0\|_2$ 
 $\rho_1 = \xi_1 = 0$ ,  $\epsilon_0 \neq 0$ ,  $\kappa_0 = -1$ 
for  $n = 1, 2, 3, \dots$  do
•  $\delta_n = \mathbf{w}_n^T \mathbf{v}_n$ 
 $\mathbf{p}_n = \mathbf{v}_n - \frac{\xi_n \delta_n}{\epsilon_{n-1}} \mathbf{p}_{n-1}$ 
 $\mathbf{q}_n = \mathbf{w}_n - \frac{\rho_n \delta_n}{\epsilon_{n-1}} \mathbf{q}_{n-1}$ 
•  $\epsilon_n = \mathbf{q}_n^T \mathbf{A} \mathbf{p}_n$ 
 $\beta_n = \frac{\epsilon_n}{\delta_n}$ 
 $\tilde{\mathbf{v}}_{n+1} = \mathbf{A} \mathbf{p}_n - \beta_n \mathbf{v}_n$ 
 $\tilde{\mathbf{w}}_{n+1} = \mathbf{A}^T \mathbf{q}_n - \beta_n \mathbf{w}_n$ 
•  $\rho_{n+1} = \|\tilde{\mathbf{v}}_{n+1}\|_2$ 
•  $\xi_{n+1} = \|\tilde{\mathbf{w}}_{n+1}\|_2$ 
 $\kappa_n = -\frac{\rho_n}{\beta_n} \kappa_{n-1}$ 
 $\mathbf{x}_n = \mathbf{x}_{n-1} + \kappa_n \mathbf{p}_n$ 
if  $(|\rho_{n+1} \kappa_n| < \text{tolerance})$  then STOP
 $\mathbf{v}_{n+1} = \frac{1}{\rho_{n+1}} \tilde{\mathbf{v}}_{n+1}$ 
 $\mathbf{w}_{n+1} = \frac{1}{\xi_{n+1}} \tilde{\mathbf{w}}_{n+1}$ 
endfor

```

---

**Algorithm 2:** A rescaled version of the biconjugate gradient method

## 5.4 A RESCALED VERSION OF THE ORIGINAL BCG

Since the biconjugate gradient method is the archetype of an entire class of iterative methods standard implementations of this process are given at numerous places [16, 6, 8, 18] to name a few. To draw a fair comparison between Alg. 1 and the original biconjugate gradient method, this section introduces a particular implementation of the original BCG that is taken as a contrast in the numerical experiments carried out in the next section.

In addition to the original system  $\mathbf{Ax} = \mathbf{b}$ , the biconjugate gradient method implicitly solves a dual system  $\mathbf{A}^T \tilde{\mathbf{x}} = \tilde{\mathbf{b}}$  giving rise to two sequences of residual vectors. For reasons of stability, there is no harm in scaling these residual vectors in order to avoid over- or underflow. Standard implementations usually do not involve the possibility to scale both of these vectors resulting in fewer inner product-like operations. The residual vectors of the biconjugate gradient method are, up to scaling, just the Lanczos vectors of the Lanczos algorithm. So, to draw a fair comparison with Alg. 1 which offers the scaling of both sequences of Lanczos vectors we derive an implementation of the biconjugate gradient method with scaling of both sequences of residual vectors as follows.

Alg. 3.2 of [10] essentially is a rescaled version of the biconjugate gradient method where the computation of the iterates is omitted. Actually, this is a Lanczos algorithm involving the possibility to scale the Lanczos vectors. We include the computation of the iterates in the same way as above, i.e., by adding two assignments of the form (5.16) and (5.17). The resulting process is depicted in Alg. 2 where, as before, the bullets mark inner product-like operations and are used to stress the data dependencies that lead to three global synchronization points per iteration. Note that this section is solely required for sketching an implementation of the original BCG that will serve for comparison in the numerical experiments below and does not contain any new material.

## 5.5 NUMERICAL EXPERIMENTS

This section presents some numerical experiments by comparing Alg. 1 with Alg. 2. Both algorithms are, in exact arithmetic, mathematically equivalent. Moreover, both offer the possibility to scale the two sequences of Lanczos vectors. The experiments were carried out on Intel's PARAGON XP/S 10 at Forschungszentrum Jülich with the OSF/1 based operating system, Rel. 1.4. The double precision FORTRAN code is translated using Portland-Group compiler, Rel. 5.0.3, with optimization level 3.

As an example consider the partial differential equation

$$Lu = f \quad \text{on} \quad \Omega = (0, 1) \times (0, 1)$$

with Dirichlet boundary condition  $u = 0$  where

$$Lu = -\Delta u - 20 \left( x \frac{\partial u}{\partial x} + y \frac{\partial u}{\partial y} \right)$$

and the right-hand side  $f$  is chosen so that the solution is given by

$$u(x, y) = \frac{1}{2} \sin(4\pi x) \sin(6\pi y) .$$

We discretize the above differential equation using second order centered differences on a  $440 \times 440$  grid with mesh size  $1/441$ , leading to a system of linear equations with unsymmetric real coefficient matrix of order 193 600 with 966 240 nonzero entries.



Simple diagonal preconditioning is used by transforming (5.8) into the equivalent system

$$\mathbf{M}\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{b} ,$$

where  $\mathbf{M} = \text{diag}(m_1, m_2, \dots, m_N)$  is the preconditioner with

$$m_i = \frac{1}{\sqrt{\sum_{j=1}^N a_{ij}^2}} , \quad i = 1, 2, \dots, N .$$

For our test runs we choose  $\mathbf{x}_0 = 0$  as initial guess to the exact solution and stop as soon as  $\|\mathbf{r}_n\|_2 < 10^{-6}$ . Smaller tolerances do not yield better approximations to the exact solution  $u$ ; the absolute difference between the approximations and the exact solution stagnates at  $9 \cdot 10^{-5}$ .

To partition the data among the processors the parallel implementation subdivides  $\Omega$  into square subdomains of equal size. Thus, each of the processors holds the data of the corresponding subdomain. Due to the local structure of the discretizing scheme, a processor has to communicate with at most 4 processors to perform a matrix-by-vector product. Note that, in Alg. 1, there are two independent matrix-by-vector products per iteration which are computed simultaneously in this implementation.

Figure 5.1 shows the convergence history of both versions of the biconjugate gradient method where the true residual norm  $\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2$  is displayed. Note that, in this numerical example for both algorithms and all numbers of processors considered, there is hardly any difference between the true residual norm  $\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2$  and  $\|\mathbf{r}_n\|_2$  where  $\mathbf{r}_n$  is the vector computed recursively in the algorithms. Thus, (5.19) gives a good estimate of the true residual norm here. Figure 5.1 demonstrates the wild oscillations in the residual norm that are typical for the biconjugate gradient method. A similar behavior of both algorithms is recognized. We remark that, varying the number of processors, the residual norm of Alg. 2 is essentially fixed whereas there are some differences in Alg. 1. The fact that highly parallel algorithms may be numerically less stable than their conventional serial counterparts has been observed for several other problems arising in linear algebra [5].

The parallel performance results are given in Fig. 5.2. These results are based on time measurements of a fixed number of iterations. The speedup given on the left-hand side of this figure is computed by taking the ratio of the parallel run time and the run time of a serial implementation. While the serial run times of both variants are almost identical there is a considerable difference concerning parallel run times. For all numbers of processors, the new parallel variant is faster than Alg. 2. The saving of run time grows with increasing number of processors; see right-hand side of Fig. 5.2. More precisely, the quantity depicted as a percentage is  $1 - T_1(p)/T_2(p)$ , where  $T_1(p)$  and  $T_2(p)$  are the run times on  $p$  processors of Alg. 1 and Alg. 2, respectively. Recognize from this figure that, for a fixed number of iterations, the new variant is approximately 24% faster than Alg. 2 on 121 processors. Note further that there is still room for using even more processors before running into saturation.

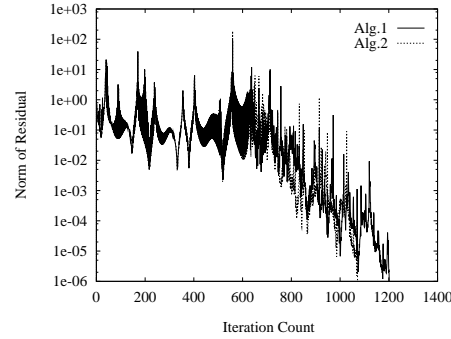


Figure 5.1 Convergence history

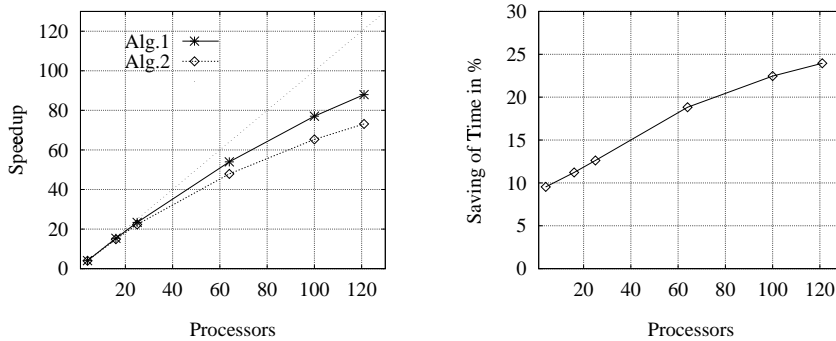


Figure 5.2 Results of the parallel implementations

## 5.6 CONCLUDING REMARKS

In its standard form, the biorthogonal Lanczos algorithm makes use of three-term recurrences for the generation of the Lanczos vectors spanning the underlying Krylov spaces. Here, we consider an alternative formulation based on coupled two-term recurrences that, in exact arithmetic, is mathematically equivalent to the three-term form. Taking a specific variant of the coupled two-term process as the starting point, a new iterative scheme for the solution of linear systems with non-Hermitian coefficient matrices is derived that, in exact arithmetic, is mathematically equivalent to the biconjugate gradient method. In this new implementation all inner product-like operations of an iteration step are independent such that the implementation consists of a single global synchronization point per iteration resulting in improved scalability on massively parallel computers. Moreover, the implementation offers the possibility to scale both of the two sequences of Lanczos vectors which is an important aspect concerning numerical stability in practical implementations.

## 5.7 ACKNOWLEDGMENTS

Over the last years, many colleagues of the Central Institute for Applied Mathematics, Forschungszentrum Jülich GmbH, gave generously of their time and competence. From each of them we received extraordinarily helpful advices. In particular, Friedel Hoßfeld, Rainer von Seggern, and Peter Weidner deserve credit for guiding us during our graduate studies, and for the hours of time and never exhausting patience that they invested in our work.

## References

- [1] Bücker, H. M. and Sauren, M. (1996a). A Parallel Version of the Quasi-Minimal Residual Method Based on Coupled Two-Term Recurrences. In Waśniewski, J., Dongarra, J., Madsen, K., and Olesen, D., editors, *Applied Parallel Computing: Industrial Computation and Optimization, Proceedings of the Third International Workshop, PARA '96, Lyngby, Denmark, August 18–21, 1996*, volume 1184 of *Lecture Notes in Computer Science*, pages 157–165, Berlin. Springer.
- [2] Bücker, H. M. and Sauren, M. (1996b). A Parallel Version of the Unsymmetric Lanczos Algorithm and its Application to QMR. Internal Report KFA-ZAM-IB-9605, Research Centre Jülich, Jülich, Germany.
- [3] Bücker, H. M. and Sauren, M. (1997). A Variant of the Biconjugate Gradient Method Suitable for Massively Parallel Computing. In Bilardi, G., Ferreira, A., Lüling, R., and Rolim, J., editors, *Solving Irregularly Structured Problems in Parallel, Proceedings of the Fourth International Symposium, IRREGULAR'97, Paderborn, Germany, June 12–13, 1997*, volume 1253 of *Lecture Notes in Computer Science*, pages 72–79, Berlin. Springer.
- [4] Cullum, J. K. and Greenbaum, A. (1996). Relations between Galerkin and Norm-Minimizing Iterative Methods for Solving Linear Systems. *SIAM Journal on Matrix Analysis and Applications*, 17(2):223–247.
- [5] Demmel, J. W. (1993). Trading Off Parallelism and Numerical Stability. In Moonen, M. S., Golub, G. H., and Moor, B. L. R. D., editors, *Linear Algebra for Large Scale and Real-Time Applications*, volume 232 of *NATO ASI Series E: Applied Sciences*, pages 49–68. Kluwer Academic Publishers, Dordrecht, The Netherlands. Proceedings of the NATO Advanced Study Institute on Linear Algebra for Large Scale and Real-Time Applications, Leuven, Belgium, August 1992.
- [6] Fletcher, R. (1976). Conjugate Gradient Methods for Indefinite Systems. In Watson, G. A., editor, *Numerical Analysis Dundee 1975*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89, Berlin. Springer.
- [7] Freund, R. W. (1993). The Look-Ahead Lanczos Process for Large Nonsymmetric Matrices and Related Algorithms. In Moonen, M. S., Golub, G. H., and Moor, B. L. R. D., editors, *Linear Algebra for Large Scale and Real-Time Applications*, volume 232 of *NATO ASI Series E: Applied Sciences*, pages 137–163. Kluwer Academic Publishers, Dordrecht, The Netherlands. Proceedings of the NATO Advanced Study

Institute on Linear Algebra for Large Scale and Real-Time Applications, Leuven, Belgium, August 1992.

- [8] Freund, R. W., Golub, G. H., and Nachtigal, N. M. (1992). Iterative Solution of Linear Systems. In *Acta Numerica 1992*, pages 1–44. Cambridge University Press, Cambridge.
- [9] Freund, R. W., Gutknecht, M. H., and Nachtigal, N. M. (1993). An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices. *SIAM Journal on Scientific Computing*, 14(1):137–158.
- [10] Freund, R. W. and Nachtigal, N. M. (1994). An Implementation of the QMR Method Based on Coupled Two-Term Recurrences. *SIAM Journal on Scientific Computing*, 15(2):313–337.
- [11] Golub, G. H. and Loan, C. F. V. (1996). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, third edition.
- [12] Gutknecht, M. H. (1992). A Completed Theory of the Unsymmetric Lanczos Process and Related Algorithms, Part I. *SIAM Journal on Matrix Analysis and Applications*, 13(2):594–639.
- [13] Gutknecht, M. H. (1994). A Completed Theory of the Unsymmetric Lanczos Process and Related Algorithms, Part II. *SIAM Journal on Matrix Analysis and Applications*, 15(1):15–58.
- [14] Kim, S. K. and Chronopoulos, A. T. (1992). An Efficient Nonsymmetric Lanczos Method on Parallel Vector Computers. *Journal of Computational and Applied Mathematics*, 42:357–374.
- [15] Lanczos, C. (1950). An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Research of the National Bureau of Standards*, 45(4):255–282.
- [16] Lanczos, C. (1952). Solutions of Systems of Linear Equations by Minimized Iterations. *Journal of Research of the National Bureau of Standards*, 49(1):33–53.
- [17] Parlett, B. N., Taylor, D. R., and Liu, Z. A. (1985). A Look-Ahead Lanczos Algorithm for Unsymmetric Matrices. *Mathematics of Computation*, 44(169):105–124.
- [18] Saad, Y. (1996). *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston.
- [19] Taylor, D. R. (1982). *Analysis of the Look Ahead Lanczos Algorithm for Unsymmetric Matrices*. Ph. D. dissertation, Department of Mathematics, University of California, Berkeley, CA.

## 6 A NEW JACOBI ORDERING FOR MULTIPLE-PORT HYPERCUBES

Dolors Royo, Miguel Valero-García and Antonio González

*Departament d'Arquitectura de Computadors  
Universitat Politècnica de Catalunya,  
c/ Jordi Girona 1-3, Campus Nord - Edifici D-6,  
E-08034 Barcelona, Spain*

{dolors,miguel,antonio}@ac.upc.es

**Abstract:** In a previous work we proposed two new Jacobi orderings for computing the eigenvalues of a symmetric matrix. The new orderings, combined with the application of the communication pipelining technique allow the exploitation of multi-port capabilities in hypercubes (sending several messages in parallel through the hypercube links). One of the proposed orderings, called *Permuted-BR*, is near optimal but only works for very large problems. The second ordering, called *Degree-4*, can be applied to any problem size but only reduces the communication cost by a factor of 4. In this paper we propose a new ordering, called *Permuted-D4*, that behaves like *Permuted-BR* ordering for big problems and like *Degree-4* for small ones.

### 6.1 INTRODUCTION

The one-sided Jacobi method for symmetric eigenvalue computation is very suited for its application on a multicomputer since it exhibits a high parallelism and potentially low communication requirements [5].

In multicomputers, the communication overhead plays an important role on the performance of any particular algorithm [1]. In this paper, we focus on multicomputers with a hypercube interconnection topology and with multiple ports per node [12]. In such scenario one may design algorithms that communicate multiple messages simultaneously through different links of the same node (communication parallelism), which may result in a significant reduction in the communication overhead. One-sided Jacobi algorithms previously proposed for hypercubes make a poor utilization of the multi-port capability because, at any given time, the information to be communicated

by every node must be sent through one (or at most two) link, constraining in this way the exploitation of communication parallelism.

In [8] a method was developed to design parallel algorithms that efficiently exploit the multi-port capability in hypercubes. The method, which is called communication pipelining, requires the specification of the original problem in the form of a CC-cube algorithm, whose properties will be described later. The method reorganizes the computation in a systematic way to introduce the appropriate level of communication parallelism in order to efficiently exploit the multi-port capability.

In a previous work [4], we showed that one-sided Jacobi computation can take the form of a CC-cube algorithm. In particular, the parallel one-sided Jacobi algorithm that uses the Block Recursive (BR) ordering is a CC-cube algorithm [4]. Thus, the communication pipelining technique proposed in [8] can be used to reduce the communication cost in multi-port hypercubes. Communication pipelining has, however, a limited impact when using the BR ordering, since only two hypercube links can be used simultaneously. In [4] we proposed two alternative Jacobi orderings which enable a better exploitation of the multi-port capability in hypercubes. In particular, the *Permuted-BR* ordering enables the use of  $d$  hypercube links in parallel (being  $d$  the dimensionality of the hypercube), when solving very large problem sizes. However, it behaves as the BR ordering for small problem sizes. On the other hand, the *Degree-4* ordering (also proposed in [4]) enables the use of 4 hypercube links in parallel, for any problem size.

In this paper we propose a new Jacobi ordering that behaves as the *Permuted-BR* ordering for large problems and as the *Degree-4* for small problem sizes. Therefore, a single Jacobi ordering can replace the two Jacobi orderings previously proposed.

## 6.2 PRELIMINARIES

In this paper we focus on efficient algorithms for eigenvalues and eigenvectors computation of symmetric matrices through the one-sided Jacobi method on multicomputers with a hypercube interconnection topology and multiple ports per node. Below we summarize what it is implied by each one of these terms. Then, we review the most relevant related work and point out the motivation for this work.

### 6.2.1 Target architecture

The target architecture on which the proposed algorithm is to be executed is a multi-port hypercube multicomputer. A hypercube multicomputer of dimension  $d$ , which is also called a  $d$ -cube multicomputer, consists of  $2^d$  processors such that they can be labelled from 0 to  $2^d - 1$  in such a way that all processors that differ in exactly one bit in the binary representation of their label are neighbors (they are connected by a link). The link that connects neighbor nodes whose labels differ in the  $i$ -th bit will be referred to as link  $i$ . The label of the link, which ranges from 0 to  $d-1$ , will be also called the dimension of the link. As an example, node 2 uses link 1 (or dimension 1) to send messages to node 0.

A configuration in which every node can send and receive a message from just one of its links at a time is called *one-port* configuration [12]. A multi-port multicomputer is distinguished by the fact that every node can simultaneously be transmitting/receiving messages from more than one link. In particular, in an *all-port* configuration every node can send and receive a message through each of its  $d$  links simultaneously.

### 6.2.2 One-sided Jacobi method

The one-sided method for symmetric eigenvalue and eigenvector computation works with two matrices:  $\bar{A}$  (initially set to the original matrix  $A$ ) and  $U$  (initially set to the identity matrix  $I$ ). In every iteration of the method one similarity transformation is applied to zero one off-diagonal element of  $A$  and its symmetric one [5]. The key feature of the one-sided method is that the computation and application of the similarity transformation that zeroes elements  $(i, j)$  and  $(j, i)$  of  $A$  uses only columns  $i$  and  $j$  of  $\bar{A}$  and  $U$ . For this reason, such a transformation will be also referred to as the pairing of columns  $i$  and  $j$ . According to this, transformations that use disjoint pairs of columns can be applied in parallel, since they do not share any data. This is what makes the one-sided Jacobi method to be very suitable for parallel implementations [5].

In the literature, the process of zeroing every off-diagonal element exactly once is called a sweep. Since a similarity transformation may fill in elements that had been zeroed in previous transformations, several sweep are required until the resulting matrix converges to a diagonal matrix. Since similarity transformations preserve the eigenvalues, the elements in the diagonal of the resulting matrix coincide with the eigenvalues of the original matrix.

If  $A$  is an  $m \times m$  matrix, a total of  $m(m-1)/2$  similarity transformations are required to complete a sweep. Assuming  $m$  even, these transformations can be organized into a maximum of  $m-1$  groups of  $m/2$  independent transformations each (transformations that involve disjoint pairs of columns). Such an organization of the sweep is called a parallel Jacobi ordering and every group of independent transformations is usually referred to as a step.

Parallel Jacobi orderings offer a good framework for the design of parallel implementations, since the computation involved in every step can be evenly distributed among the nodes of the multicomputer, and carried out without any communication. Communication is required however at the end of every step, when the columns of matrices  $A$  and  $U$  must be exchanged to obtain the pairs required for the next step. Such a communication will be called a transition. A parallel Jacobi ordering is suitable for a multicomputer if the transitions between steps use a communication pattern that matches the topology of the interconnection network of the machine.

### 6.2.3 Related work

Jacobi orderings for different parallel architectures can be found in the literature [3] [6][13][11]. Regarding hypercube multicomputers, which is the target architecture considered in this work, the most relevant proposals are [2][7][10].

All these proposals share the following features: (a) they use a minimum number of steps per sweep, achieving in this way a perfect load balance, and (b) the transitions between steps can be implemented through communication between neighbors in the hypercube. Particularly relevant to our work is the Block Recursive (BR) ordering proposed in [7] and reviewed in [1]. The distinguishing feature of this ordering is that in every transition all nodes exchange information through the same dimension of the hypercube. This is one of the requirements to enable the application of the communication pipelining technique, which increases the potential communication parallelism of the algorithms, and therefore facilitates the exploitation of the multi-processor feature. In the following we describe the BR ordering in some detail, since it is the starting point for our proposals.

**6.2.3.1 BR ordering.** The BR ordering was originally proposed in [7] and reviewed later in [10], where the ordering was completely specified and its correctness was shown.

Given a  $d$ -cube and a  $m \times m$  matrix, the  $m$  columns of matrices  $\bar{A}$  and  $U$  are grouped into  $2^{d+1}$  blocks of  $m/2^{d+1}$  columns each. Then, a pair of blocks are allocated to each node of the  $d$ -cube. The algorithm to perform the first sweep proceeds as follows (all the nodes are executing the following code in parallel):

- (1) Pair each column of a block with the remaining columns of the same block.
- (2) Pair each column of a block with all the columns of the other block allocated to the same processor.
- (3) Exchange one of the blocks with a neighbor along a given dimension of the hypercube (transition).
- (4) Go to (2) if not all the columns have been paired (i.e., (2) and (3) are repeated  $2^{d+1} - 1$  times).

In order to achieve that every pair of blocks is paired exactly once, the transitions (i.e. the exchange operations required by (3)) must be performed in a particular order. To precisely define a exchange transition it suffices to identify the link of the hypercube (i.e., the dimension) that is used by it. Below we describe the sequence of transitions implied by the BR ordering as described in [10].

The  $2^{d+1} - 1$  steps of the ordering are grouped into three different types of *phases*: *exchange*, *division* and *last transition*. A phase is just one or more consecutive steps, each of them followed by a transition. A sweep consists of  $d$  exchange phases, each one of them followed by a division phase, and a final last transition phase. The exchange phases are numbered from  $d$  to 1. The exchange phase  $e$  ( $e \in [d, 1]$ ) consists of  $2^e - 1$  steps and transitions. The sequence of links (i.e. dimensions) that define the transitions is denoted by  $D_e^{BR}$  and can be systematically generated as follows:

$$\begin{aligned} D_1^{BR} &= \langle 0 \rangle \\ D_i^{BR} &= \langle D_{i-1}^{BR}, i-1, D_{i-1}^{BR} \rangle \quad 1 < i \leq e \end{aligned}$$



For instance, the sequence of links for  $e=4$  is  $D_4^{BR} = \langle 010201030102010 \rangle$ .

The exchange phase  $e$  is followed by a division phase that consists of just one step and one transition through link  $e$ . Finally, the last transition phase consists of one step and one transition through link  $d-1$ .

The second and next sweeps use the same algorithm but after applying a permutation to the links. In particular, the permutation corresponding to sweep  $\sigma$  (assuming  $\sigma=0$  for the first one) is defined as follows:

$$\begin{aligned}\sigma_0(i) &= i \\ \sigma_s(i) &= (\sigma_{s-1}(i) - 1) \bmod d \quad \text{for } i=0\dots d-1\end{aligned}$$

After  $d$  sweeps the links are used again in the order described for the first sweep.

## 6.2.4 Motivation

The previously summarized BR algorithm is efficient for a *one - port* hypercube since the system is using all the available ports for every transition. However, for a multi-port architecture it achieves the same performance as for a *one - port* because it uses just one out of the  $d$  links of each node at the same time.

In [8] we proposed a systematic transformation of a class of algorithms that is called CC-cube algorithms in order to optimize them for a multi-port environment. Such transformation is called *communication pipelining*. Every process of the CC-cube algorithm executes a loop that iterates  $K$  times. Every iteration consists of a certain computation and an exchange of information through one of the hypercube dimensions (all the processes use the same dimension in a given iteration of the loop). The key idea of communication pipelining is to decompose the computation in every iteration into  $Q$  packets and reorganize the computation as follows. Every process of the new algorithm (the pipelined CC-cube algorithm) compute the first packet of the computation in the first iteration of the original CC-cube and exchanges the result with the corresponding neighbor. Then, it performs the second packet of the first iteration and the first packet of the second one. The results of these packets can be sent in parallel to the corresponding neighbors, using simultaneously two hypercube dimensions. Proceeding in this way, the pipelined CC-cube can send an increasing number of messages in parallel, being able to exploit the multi-port feature.

The value of  $Q$  is called *pipelining degree* and in [8] it is shown how to determine the pipelining degree that minimizes the execution time of any particular CC-cube algorithm and any particular hypercube architecture. In this section we only summarize the result of the application of the communication pipelining technique and omit any detail that is not strictly necessary to understand this paper. The interested reader is referred to the original paper for such details.

If  $Q$  is not higher than  $K$ , the pipelined CC-cube algorithm is said to work in shallow pipelining mode. Every process can send up to  $Q$  messages simultaneously

through different links if the architecture supports it. These  $Q$  links correspond to all the subsequences of  $Q$  elements of the sequence that describes the link usage in the CC-cube algorithm.

For instance, if in the original CC-cube  $K=7$  and communications are carried out through links 0, 1, 0, 2, 0, 1 and 0, the pipelined CC-cube with  $Q=3$  will perform a series of computations each one followed by communication through links 0-1-0, 1-0-2, 0-2-0 and 2-0-1 respectively. This part of the pipelined CC-cube is called the kernel and has  $K-Q$  stages (computation followed by communication).

Like in the software pipelining technique [9], the kernel is preceded by a prologue and followed by an epilogue. The prologue consists of  $Q-1$  stages and each of these stages consists of a computation followed by a communication through an increasing number of the first links of the original sequence. In the previous example, the prologue consists of two stages that use the following links: 0 and 0-1 respectively. Similarly, the epilogue consists of  $Q-1$  stages and each of these stages consists of a computation followed by a communication through a decreasing number of the last links of the original sequence. In the previous example, the epilogue consists of two stages that use the following links: 1-0 and 0.

If  $Q$  is higher than  $K$ , the pipelined CC-cube is said to work in deep pipelining mode. In this case the prologue and epilogue have  $K-1$  stages each and the kernel has  $Q-K+1$  stages. In each kernel stage,  $K$  messages can be sent in parallel (one packet from every iteration of the original CC-cube). For instance, if  $K=3$  and the links are used in the order 0, 1, and 0, the pipelined CC-cube with  $Q=100$  has a prologue with two stages that use links 0 and 0-1. Every one of the 98 stages of the kernel use links 0-1-0. Finally, the two stages of the epilogue use the links 1-0 and 0.

Communication pipelining can be applied to any CC-cube algorithm that meets some requirements as specified in [8]. In particular, it can be applied to every exchange phase of BR algorithm, which are the most time-consuming part of it. The application of communication pipelining to the BR algorithm can reduce the communication overhead by a factor not higher than 2. This is basically due to the properties of the BR ordering: in the sequence  $D_e^{BR}$ , which defines the order in which the links of each node are used in exchange phase  $e$ , any subsequence of  $Q$  consecutive elements has at least  $\lfloor Q/2 \rfloor$  elements equal to 0. In consequence, the capability to send simultaneously messages through multiple links can provide a reduction in the communication cost by a factor not higher than 2, since about half of the messages must be sent through the same link (link 0).

From the point of view of effective communication pipelining, we would like a Jacobi ordering where sequences  $D_e$ , defining the order in which the hypercube links are used in exchange phase  $e$ , have the following properties:

- (a) All the hypercube links appear the same number of times in the sequence  $D_e$ .
- (b) When taking any subsequence of consecutive links in  $D_e$ , all the hypercube links appear in this subsequence the same number of times.

Property (a) guarantees an optimal performance when working in deep pipelining mode, since all the data to be exchanged in every stage is well distributed among all the hypercube links. Property (b) guarantees that a good data distribution among hypercube links also happens when working in shallow pipelining mode.

The *Permuted-BR* ordering proposed in [4] is near optimal with regard to property (a). It was obtained by applying a systematic transformation to the  $D_e^{BR}$  sequence. The new sequence was denoted by  $D_e^{p-BR}$ . The aim of the systematic transformation is to balance the use of the hypercube links when considering the whole sequence. However, the distribution of the hypercube links when considering small subsequences of  $D_e^{p-BR}$  remains similar to the case of the  $D_e^{BR}$  sequence. As a result, the *Permuted-BR* ordering is near optimal when working in deep pipelining mode but it behaves as the BR ordering when working in shallow pipelining mode. Since deep pipelining mode requires very large problem sizes, the *Permuted-BR* ordering is only suitable for very large problem sizes.

The second Jacobi ordering proposed in [4], which is called *Degree-4* ordering, was conceived with property (b) in mind. The sequence for the exchange phases in this ordering is denoted by  $D_e^{D4}$ . The main feature of this sequence is that all the links in almost every subsequence of 4 consecutive links in  $D_e^{D4}$  are different. Therefore, 4 hypercube links can be used in parallel in every stage, when working in shallow pipelining mode, with  $Q=4$ .

The key idea in this paper is to apply the systematic transformation that was used in [4] to derive the  $D_e^{p-BR}$  sequence, but using the  $D_e^{D4}$  sequence as starting point, instead of the  $D_e^{BR}$  sequence. The resulting sequence, which will be called  $D_e^{p-D4}$ , will behave as the  $D_e^{p-BR}$  sequence in deep pipelining mode and as the  $D_e^{p-D4}$  sequence in shallow pipelining mode with  $Q=4$ .

In the following we give some details on the *Degree-4* ordering and on the systematic transformation to be used to derive the  $D_e^{p-D4}$  sequence.

### 6.3 THE DEGREE-4 ORDERING

The *Degree-4* ordering is obtained from the BR ordering by replacing the sequence  $D_e^{BR}$  ( $e \in [d, 1]$ ) by an alternative sequence that, besides generating all the required pairings of columns, exhibit a more balanced use of the links of every node, enabling a more efficient exploitation of the multi-port capability.

The sequence  $D_e^{BR}$  can be regarded as the definition of a hamiltonian path in an  $e$ -cube. This is because one half of the columns of A and U initially allocated to each node must visit a different node after each transition and thus, they must have visited all the nodes after the  $2^e - 1$  transitions that make up exchange phase  $e$ . For instance, the exchange phase  $e = 3$  of the BR ordering is defined by  $D_3^{BR} = \langle 0102010 \rangle$ . Starting at any node of the hypercube and moving through the links in the order given in  $D_3^{BR}$ , all the nodes of a 3-cube are visited exactly once.

We can then conclude that the problem of defining alternative sequences  $D_e$  can also be stated as the problem of finding alternative hamiltonian paths in an  $e$ -cube. The

resulting sequence consists of the link identifiers that have been used for traversing the  $e$ -cube. Since there are many different hamiltonian paths in a hypercube, there are many alternative Jacobi orderings that can be generated in this way.

The *Degree-4* ordering proposed in [4] uses the  $D_e^{D^4}$  sequence defined as follows:

$$\begin{aligned} E_3 &= \langle 0123012 \rangle \\ E_i &= \langle E_{i-1}, i, E_{i-1} \rangle & 4 \leq i < e \\ D_e^{D^4} &= \langle E_{e-1}, 1, E_{e-1} \rangle & e \geq 4 \end{aligned}$$

For instance,  $D_5^{D^4} = \langle 012301240123012101230120123012 \rangle$ . In [4] it is shown that  $D_e^{D^4}$  is a hamiltonian path of a  $e$ -cube. Note that all the subsequences of length 4 in sequence  $D_e^{D^4}$  use 4 different hypercube links, with the only exception of the four central subsequences ( $\langle 0121 \rangle$ ,  $\langle 1210 \rangle$ ,  $\langle 2101 \rangle$  and  $\langle 1012 \rangle$  in the previous example). This is true for any  $e > 3$ . When  $e$  is large, it can be shown that these 4 central subsequences have a negligible effect on the performance. As a result, one-sided algorithms that use the *Degree-4* ordering and the communication pipelining technique (with  $Q=4$ ) have a communication cost 4 times lower than the BR algorithm.

## 6.4 A SYSTEMATIC TRANSFORMATION OF HAMILTONIAN PATHS

To be effective when working in deep pipelining mode, the sequence  $D_e$  must satisfy as much as possible property (a) described in section 2.4. Let us define  $\alpha$  as the number of repetitions of the most frequent dimension in a sequence  $D_e$ . For instance, the value of  $\alpha$  in the case of  $D_e^{BR}$  is  $2^{e-1}$  which is the number of repetitions of dimension 0, and the value of  $\alpha$  in the case of  $D_e^{p-D^4}$  is  $2^{e-2} + 1$  which is the number of repetitions of dimension 1. The value of  $\alpha$  is a good measure of the quality of a sequence  $D_e$  with regard to property (a). Good sequences have low values of  $\alpha$ . In particular, a lower bound for the value of  $\alpha$  of any sequence  $D_e$  is:

$$\left\lfloor \frac{2^e - 1}{e} \right\rfloor$$

since any sequence  $D_e$  has  $2^e - 1$  elements and all values in  $[0, e - 1]$  must appear in  $D_e$  at least once.

The transformation that we use to generate  $D_e^{p-D^4}$  is aimed at reducing the value of  $\alpha$  corresponding to the original sequence  $D_e^{D^4}$ . The transformation is based on the following property:

**Property.** Let  $D_e$  be a sequence of links that define a hamiltonian path in an  $e$ -cube. Let  $\sigma$  be any permutation of the link identifiers. If we apply  $\sigma$  to  $D_e$  then the resulting sequence defines also a hamiltonian path of the  $e$ -cube. This property is proved in [4].

$$\begin{aligned}
D_6^{D^4} &= \langle 0123012 \text{ 4 } \mathbf{0123012} \text{ 5 } 0123012 \text{ 4 } \mathbf{0123012} \text{ 1 } \\
&\quad \mathbf{0123012} \text{ 4 } 0123012 \text{ 5 } \mathbf{0123012} \text{ 4 } 0123012 \rangle \\
&\quad \text{(a)} \\
D_6^{p-D^4} &= \langle 0123012 \text{ 4 } 0123012 \text{ 5 } \mathbf{4321432} \text{ 0 } \mathbf{4321432} \text{ 3 } \\
&\quad \mathbf{4321432} \text{ 0 } \mathbf{4321432} \text{ 5 } 0123012 \text{ 4 } 0123012 \rangle \\
&\quad \text{(b)}
\end{aligned}$$

Figure 6.1 (a) The four 4-sequences in  $D_6^{D^4}$  are shown in different font types. Links 4 and 5 separate the different 4-sequences. (b) Transformation  $T_0$  affects to the second 5-sequence. Links affected are shown in bold.

As an example, consider the sequence  $\langle 0102010 \rangle$ , which defines a hamiltonian path of a 3-cube. The permutation that transposes links 0 and 1 in this sequence would produce the new sequence  $\langle 1012101 \rangle$ , which is also a hamiltonian path of a 3-cube. Based on this property, we define the transformation  $T_k$  to be applied to a sequence  $D_e$ , as the transposition of the following pairs of link identifiers:

$$\begin{aligned}
i &\leftrightarrow \lfloor (e-1)/2^k \rfloor - 1 - i \\
\text{where } i &\in [0, \lfloor (e-1)/2^k \rfloor - 1]
\end{aligned}$$

In the following section we show how transformation  $T_k$  is applied to  $D_e^{D^4}$  to obtain the new sequence  $D_e^{p-D^4}$ .

## 6.5 PERMUTED-D4 ORDERING

The *Permuted-D4* ordering proposed in this paper uses the sequence  $D_e^{p-D^4}$  to implement the exchange phase  $e$ . The sequence  $D_e^{p-D^4}$  is obtained by applying a series of transformations  $T_k$ , as defined above, to the  $D_e^{D^4}$  sequence.

Every transformation  $T_k$  is applied to a part of  $D_e^{D^4}$  defining a hamiltonian path of a subcube of dimension  $(e-k-1)$  in the  $e$ -cube. Such a part of  $D_e^{D^4}$  will be referred to as a  $k$ -sequence. As a result of the property introduced in section 4, the result of such a transformation is still a hamiltonian path of a  $e$ -cube. We have to specify now what transformations are used to produce  $D_e^{p-D^4}$ .

Notice that in  $D_e^{D^4}$  we can find two  $(e-1)$ -sequences, four  $(e-2)$ -sequences and, in general  $2^k$   $(e-k)$ -sequences. The links that define those  $k$ -sequences do not appear contiguous in  $D_e^{D^4}$ . For each  $k$ -sequence, one half of the links appears in the left-hand side of  $D_e^{D^4}$  and the other half appears in the symmetric position in the right-hand side of  $D_e^{D^4}$ . Figure 6.1(a) identifies in different font types the four 4-sequences in  $D_6^{D^4}$ .

$\lfloor \log_2(e-1) \rfloor - 1$  transformations are applied to  $D_e^{D^4}$  in order to obtain  $D_e^{p-D^4}$ . Transformation  $T_k$ ,  $k$  being an integer from 0 to  $\lfloor \log_2(e-1) \rfloor - 2$ , is applied to every second  $(e-k-1)$ -sequence of  $D_e^{D^4}$ . In the case of  $D_6^{D^4}$  only one transformation  $T_0$  is required which is applied to the second 5-sequence. Figure 6.1(b) shows in boldface the links that are affected by the transformation.

$e$	$D_e^{p-BR}$	Lower bound	$D_e^{p-D4}$
7	23	19	24
8	43	32	40
9	67	58	72
10	131	103	136
11	289	187	264
12	577	342	520
13	776	631	784
14	1543	1171	1544

Figure 6.2 Value of  $\alpha$  corresponding to Permuted-D4 ordering compared to lower bound and Permuted-BR.

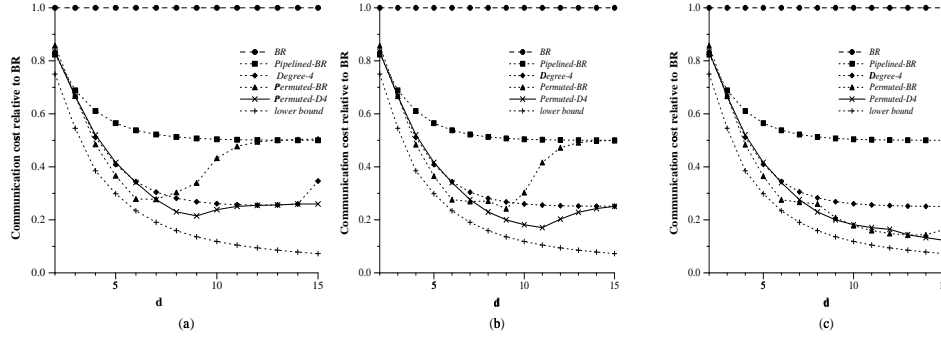


Figure 6.3 Plot (a) matrix size  $2^{18}$ . Plot (b) matrix size  $2^{23}$  and plot (c) matrix size  $2^{32}$ .

There is one point to remark, regarding the application of the transformations. When two or more transformations are required then these transformations must be compounded. That is, if a transformation requires the transposition  $i \leftrightarrow j$  in a given  $k$ -sequence, then this transposition affects to the links in the  $k$ -sequence that were labelled as  $i$  or  $j$  in the original sequence  $D_e^{D4}$ , even though these links may have changed the label as a result of previous transformations also affecting the  $k$ -sequence.

The table of figure 6.2 shows the value of  $\alpha$  for sequences  $D_e^{p-D4}$  with  $e \in [7, 14]$  and compares these values with the corresponding values of  $\alpha$  for sequences  $D_e^{p-BR}$  and with the lower bound. From the values in the table it can be concluded that the behaviour of  $D_e^{p-D4}$  is nearly the same as the behaviour of  $D_e^{p-BR}$  in deep pipelining mode, because the value of  $\alpha$  for the  $D_e^{p-D4}$  sequence is close to the value of  $\alpha$  for  $D_e^{p-BR}$ , and sometimes lower.

## 6.6 PERFORMANCE EVALUATION

In the previous section we have argued that the performance of the *Permuted-D4* ordering is nearly optimal in deep pipelining mode and can reduce the communication cost by a factor near to 4 in shallow pipelining mode. We confirm these conclusions in this section with some performance figures obtained through analytical models of performance.

The plots in figure 6.3 show the communication cost of the different orderings for a varying size of a hypercube multicomputer; a varying matrix size; and assuming that the transmission time per element and the start-up time,  $T_w$  and  $T_s$ , are 1 and 1000 time units respectively. The communication cost has been computed through the models developed in [8]. The communication cost is given in relation to the cost of the CC-cube algorithm using the BR ordering. As a reference, all plots also include the communication costs of *Permuted-BR* ordering, *Degree-4* ordering and the lower bound on the communication cost.

When communication pipelining is used, the optimum degree of pipelining was chosen, using the procedure presented in [8] for each particular hypercube dimension and matrix size.

It can be seen in all plots that the communication cost of the pipelined CC-cube when the BR ordering is used is about one half of that of the original CC-cube. The performance of the *Permuted-D4* ordering approaches the lower bound when deep pipelining is used. However, when the hypercube size forces the use of shallow pipelining, it tends to be similar to the *Degree-4* ordering, in contrast with *Permuted-BR*, which tends to be similar to the pipelined BR.

## 6.7 CONCLUSIONS

We have proposed one novel Jacobi ordering called *Permuted-D4* ordering obtained by applying the systematic transformations that were used to generate the *Permuted-BR*. The result is an ordering that makes a nearly balanced use of all hypercube dimensions and try to distributed the dimensions as well as *Degree-4* ordering.

This results in a performance close to the optimum when deep pipelining mode can be applied. In case that only shallow pipelining mode can be applied it behaves as well as *Degree-4* ordering, dividing the communication cost by 4 respect the communication cost of BR ordering.

## References

- [1] W.C. Athas and C.L. Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer*, 99(7):9–24, August 1988.
- [2] C. Bischof. The two-sided jacobi method on a hypercube. In *SIAM Proceedings of the Second Conference on Hypercube Multiprocessors*, August 1987.

- [3] R.P. Brent and F.T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Statist. Comput.*, (6):69–84, August 1985.
- [4] A. González D. Royo and M. Valero-García. Jacobi orderings for multi-port hypercubes. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, pages 88–98, 1998.
- [5] P.J. Eberlein. On one-sided jacobi methods for parallel computation. *SIAM J. Algebraic Discrete Methods*, (8):790–796, 1987.
- [6] P.J. Eberlein and H. Park. Efficient implementation of jacobi algorithms and jacobi sets on distributed memory architectures. *Journal of Parallel Distributed Computing*, (8):358–366, August 1990.
- [7] G.R. Gao and S.J. Thomas. An optimal parallel jacobi-like solution method for singular value decomposition. In *Proceedings of Int'l Conf. on Parallel Processing*, pages 47–53, 1988.
- [8] A. González L. Díaz de Cerio and M. Valero-García. Communication pipelining in hypercubes. *Parallel Processing Letters*, 6(4), December 1996.
- [9] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Conf. on Programming Language, Design and Implementation*, pages 318–328, 1988.
- [10] M. Mantharam and P.J. Eberlein. Block-recursive algorithm to generate jacobi-sets. *Parallel Computing*, (19):481–496, 1993.
- [11] M. Mantharam and P.J. Eberlein. New jacobi-sets for parallel computation. *Parallel Computing*, (19):437–454, 1993.
- [12] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in directed networks. *IEEE Computer*, 26(2):62–76, February 1993.
- [13] J. Gotze S. Paul and M. Sauer. An efficient jacobi-like algorithm for parallel eigenvalue computation. *IEEE Transactions on Computers*, C42(9):1058–1063, 1993.



## 7 THEORETICAL PERFORMANCE ANALYSIS OF THE IQMR METHOD ON DISTRIBUTED MEMORY COMPUTERS

Tianruo Yang,

tiaya@ieee.org

Hai-Xiang Lin

*Department of Technical Mathematics and Computer Science  
TU Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands.*

h.x.lin@twi.tudelft.nl

**Abstract:** For the solutions of linear systems of equations with unsymmetric coefficient matrices, we have proposed an improved version of the quasi-minimal residual (IQMR) method [14] by using the Lanczos process as a major component combining elements of numerical stability and parallel algorithm design. The algorithm is derived such that all inner products and matrix-vector multiplications of a single iteration step are independent and communication time required for inner product can be overlapped efficiently with computation time. In this paper, we mainly present the qualitative analysis of the parallel performance with *Store-and-Forward routing* and *Cut-Through routing* schemes and topologies such as *ring*, *mesh*, *hypercube* and *balanced binary tree*. Theoretically it is shown that the hypercube topology can give us the best parallel performance with regards to parallel efficiency, speed-up, and runtime, respectively. We also study theoretical aspects of the overlapping effect in the algorithm.

## 7.1 INTRODUCTION

One of the fundamental task of numerical computing is the ability to solve linear systems with nonsymmetric coefficients. These systems arise very frequently in scientific computing, for example from finite difference or finite element approximations to partial differential equations, as intermediate steps in computing the solution of nonlinear problems or as subproblems in linear and nonlinear programming.

One of them, the quasi-minimal residual (QMR) algorithm [8], uses the Lanczos process [7] with look-ahead, a technique developed to prevent the process from breaking down in case of numerical instabilities, and in addition imposes a quasi-minimization principle. This combination leads to a quite efficient algorithm among the most frequently and successfully used iterative methods. This method are widely used for very large and sparse problems, which in turn are often solved on massively parallel computers.

On massively parallel computers, the basic time-consuming computational kernels of QMR are usually: inner products, vector updates, matrix-vector multiplications. In many situations, especially when matrix operations are well-structured, these operations are suitable for implementation on vector and share memory parallel computers [6]. But for parallel distributed memory machines, the matrices and vectors are distributed over the processors, so that even when the matrix operations can be implemented efficiently by parallel operations, we still can not avoid the global communication, i.e. communication of all processors, required for inner product computations. Vector updates are perfectly parallelizable and, for large sparse matrices, matrix-vector multiplications can be implemented with communication between only nearby processors. The bottleneck is usually due to inner products enforcing global communication. The detailed discussions on the communication problem on distributed memory systems can be found in [3, 5]. These global communication costs become relatively more and more important when the number of the parallel processors is increased and thus they have the potential to affect the scalability of the algorithm in a negative way [3, 5].

Recently, we have proposed a new improved two-term recurrences Lanczos process [14] without look-ahead as the underlying process of QMR. The algorithm is reorganized without changing the numerical stability so that all inner products and matrix-vector multiplications of a single iteration step are independent and communication time required for inner product can be overlapped efficiently with computation time. Therefore, the cost of global communication on parallel distributed memory computers can be significantly reduced. The resulting IQMR algorithm maintains the favorable properties of the Lanczos process while not increasing computational costs. In [13], we also propose a theoretical model of computation and communications phases to allow us to give a qualitative analysis of the parallel performance on a massively distributed memory computer with two-dimensional grid topology. The efficiency, speed-up, and runtime are expressed as functions of the number of processors scaled by the number of processors that gives the minimal runtime for the given problem size. This provides a natural way to analyze the performance characteristics for the range of the number of processors that can be used effectively. The model not only shows clearly the dramatic

influence of global communication on the performance, but also evaluates effectively the improvements in the performance due to the communication reductions by overlapping. The model also provides useful insight in the scalability of IQMR method. But it is still limited by assumptions on the communication model. In this paper, we mainly investigate the effect of *Store-and-Forward routing* and *Cut-Through routing* and topologies such as *ring*, *mesh*, *hypercube* and *balanced binary tree*. Theoretically it is shown that the hypercube topology can give us the best parallel performance with regards to parallel efficiency, speed-up, and runtime, respectively.

The paper is organized as follows. We will describe briefly the improved Lanczos process and the resulting improved quasi-minimal residual (IQMR) method in section 7.2, 7.3 and 7.4, respectively. In section 7.5, the parallel performance model is presented including the communication model and assumptions for computation time and communication costs. The theoretical complexity analysis of parallel performance with *Store-and-Forward routing* and *Cut-Through routing* and different topologies such as *ring*, *mesh*, *hypercube* and *balanced binary tree*, is described fully in section 7.6. Finally we offer some conclusions.

## 7.2 LANCZOS PROCESS BASED ON THREE-TERM RECURRENCE

The classical unsymmetric Lanczos process [12] based on three-term recurrence reduces a matrix  $A$  to a tridiagonal form  $T$  using a similarity transformation which leads to the following three relations that serve to derive the unsymmetric Lanczos process :

$$W^T V = I, \quad AV = VT, \quad A^T W = WT^T, \quad (7.1)$$

where  $I$  is the identity matrix and  $T$  is a tridiagonal matrix. More precisely, this process, starting with two vectors  $v_1$  and  $w_1$  satisfying  $w_1^T v_1 = 1$ , iteratively generates two finite sequences of vectors  $v_n$  and  $w_n$  such that, for  $n = 1, 2, \dots$

$$\mathcal{K}_k(v_1, A) = \text{span}\{v_1, v_2, \dots, v_n\}, \quad \mathcal{K}_k(w_1, A^T) = \text{span}\{w_1, w_2, \dots, w_n\},$$

and the two sets are biorthogonal as follows

$$w_m^T v_n = \begin{cases} 0 & m \neq n, \\ 1 & m = n \end{cases}$$

We denote by

$$V = [v_1, v_2, \dots, v_n] \quad \text{and} \quad W = [w_1, w_2, \dots, w_n],$$

the matrices containing the Lanczos vectors  $v_n$  and  $w_n$  as columns.

This process leads to two inner products per iteration. The inner product requires global communication on parallel distributed memory computers. Some improvements on the reduction of global communication required by inner products have been investigated in [10].

### 7.3 LANCZOS PROCESS BASED ON COUPLED TWO-TERM RECURRENCE

Although Lanczos used a similar technique built on coupled two-term recurrence in the early of 1950's, most published papers have been dealing with the three-term recurrence process, until, recently, Freund et al. [9] reused this idea to improve numerical stability. They claimed that, the latter variant of the Lanczos process, may be numerically more stable. That is why we pursue further on this unsymmetric Lanczos process with two-term recurrences as underlying process of the QMR method.

Recently, Bücker et al. [1, 2] proposed a new parallel version of the QMR method based on the coupled two-term recurrences Lanczos process without look-ahead strategy. The algorithm is derived such that both sequences of generated Lanczos vectors are scalable and there is only one single global synchronization point per iteration. Based on the similar idea, we present a new improved two-term recurrences Lanczos process without look-ahead technique.

---

**Algorithm 1:** Improved Lanczos Process

```

1:  $p_0 = q_0 = u_0 = 0, \gamma_1 = (\tilde{v}_1, \tilde{v}_1), \mu_1 = 0, \xi_1 = (\tilde{w}_1, \tilde{w}_1),$ 
2:  $s_1 = A^T \tilde{w}_1, \rho_1 = (\tilde{w}_1, \tilde{v}_1), \varepsilon_1 = (s_1, \tilde{v}_1), \tau_1 = \frac{\varepsilon_1}{\rho_1};$ 
3: for  $n = 1, 2, \dots$  do
4:    $q_n = \frac{1}{\xi_n} s_n - \frac{\gamma_n \mu_n}{\xi_n} q_{n-1};$ 
5:    $\tilde{w}_{n+1} = q_n - \frac{\tau_n}{\xi_n} \tilde{w}_n;$ 
6:    $s_{n+1} = A^T \tilde{w}_{n+1};$ 
7:    $t_n = A \tilde{v}_n;$ 
8:    $u_n = \frac{1}{\gamma_n} t_n - \mu_n u_{n-1};$ 
9:    $\tilde{v}_{n+1} = u_n - \frac{\tau_n}{\gamma_n} \tilde{v}_n;$ 
10:   $p_n = \frac{1}{\gamma_n} \tilde{v}_n - \mu_n p_{n-1};$ 
11:   $\gamma_{n+1} = (\tilde{v}_{n+1}, \tilde{v}_{n+1});$ 
12:   $\xi_{n+1} = (\tilde{w}_{n+1}, \tilde{w}_{n+1});$ 
13:   $\rho_{n+1} = (\tilde{w}_{n+1}, \tilde{v}_{n+1});$ 
14:   $\varepsilon_{n+1} = (s_{n+1}, \tilde{v}_{n+1});$ 
15:   $\mu_{n+1} = \frac{\gamma_n \xi_n \rho_{n+1}}{\gamma_{n+1} \tau_n \rho_n};$ 
16:   $\tau_{n+1} = \frac{\varepsilon_{n+1}}{\rho_{n+1}} - \gamma_{n+1} \mu_{n+1};$ 
17: end for
```

---

We reschedule the computation within an iteration step, without affecting the numerical stability, such that all inner products and matrix-vector multiplications of a single iteration step are independent and the communication time required for inner product can be overlapped efficiently with computation time. The framework of this improved Lanczos process based on two-term recurrences is described in Algorithm 1.

The improved Lanczos process can be efficiently parallelized as follows:

- The inner products of a single iteration step (11), (12), (13) and (14) are independent.
- The matrix-vector multiplications of a single iteration step (6) and (7) are independent.
- The communications required for the inner products (11), (12), (13) and (14) can be overlapped with the update for  $p_n$  in (10).

Therefore, the cost of communication time on parallel distributed memory computers can be significantly reduced.

## 7.4 THE IMPROVED QUASI-MINIMAL RESIDUAL METHOD

The improved Lanczos process now is used as a major component to a Krylov subspace method for solving a system of linear equations

$$Ax = b, \quad \text{where } A \in \mathbb{R}^{n \times n} \quad \text{and} \quad x, b \in \mathbb{R}^n. \quad (7.2)$$

In each step, it produces approximation  $x_n$  to the exact solution of the form

$$x_n = x_0 + \mathcal{K}_n(r_0, A), \quad n = 1, 2, \dots \quad (7.3)$$

Here  $x_0$  is any initial guess for the solution of linear systems,  $r_0 = b - Ax_0$  is the initial residual, and  $\mathcal{K}_n(r_0, A) = \text{span}\{r_0, Ar_0, \dots, A^{n-1}r_0\}$ , is the  $n$ -th Krylov subspace with respect to  $r_0$  and  $A$ .

Given any initial guess  $x_0$ , the  $n$ -th Improved QMR iterate is of the form

$$x_n = x_0 + V_n z_n, \quad (7.4)$$

where  $V_n$  is generated by the improved Lanczos process, and  $z_n$  is determined by a quasi-minimal residual property.

Based on the similar idea in [1, 2] with computation rearrangement, we derive an improved QMR method (IQMR) based on coupled two-term recurrences with scaling of both sequence of Lanczos vectors for maintaining the numerical stability. All inner products and matrix-vector multiplications of a single iteration step in the IQMR are independent and communication time required for inner product can be overlapped efficiently with computation time. The framework of this improved QMR method using Lanczos algorithm based on two-term recurrences as underlying process is depicted in Algorithm 2.

---

### Algorithm 2: Improved Quasi-Minimal Residual Method

- 1:  $\tilde{v}_1 = \tilde{w}_1 = r_0 = b - Ax_0, \lambda_1 = 1, \kappa_0 = 1, \mu_1 = 0,$
- 2:  $p_0 = q_0 = u_0 = d_0 = f_0 = 0, \quad \gamma_1 = (\tilde{v}_1, \tilde{v}_1), \quad \xi_1 = (\tilde{w}_1, \tilde{w}_1),$

```

3:  $s_1 = A^T \tilde{w}_1, \rho_1 = (\tilde{w}_1, \tilde{v}_1), \varepsilon_1 = (s_1, \tilde{v}_1), \tau_1 = \frac{\varepsilon_1}{\rho_1};$ 
4: for  $n=1, 2, \dots$  do
5:    $q_n = \frac{1}{\xi_n} s_n - \frac{\gamma_n \mu_n}{\xi_n} q_{n-1};$ 
6:    $\tilde{w}_{n+1} = q_n - \frac{\tau_n}{\xi_n} \tilde{w}_n;$ 
7:    $s_{n+1} = A^T \tilde{w}_{n+1};$ 
8:    $t_n = A \tilde{v}_n;$ 
9:    $u_n = \frac{1}{\gamma_n} t_n - \mu_n u_{n-1};$ 
10:   $\tilde{v}_{n+1} = u_n - \frac{\tau_n}{\gamma_n} \tilde{v}_n;$ 
11:   $p_n = \frac{1}{\gamma_n} \tilde{v}_n - \mu_n p_{n-1};$ 
12:  if  $(r_{n-1}, r_{n-1}) < \text{tol}$  then
13:    quit
14:  else
15:     $\gamma_{n+1} = (\tilde{v}_{n+1}, \tilde{v}_{n+1});$ 
16:     $\xi_{n+1} = (\tilde{w}_{n+1}, \tilde{w}_{n+1});$ 
17:     $\rho_{n+1} = (\tilde{w}_{n+1}, \tilde{v}_{n+1});$ 
18:     $\varepsilon_{n+1} = (s_{n+1}, \tilde{v}_{n+1});$ 
19:     $\mu_{n+1} = \frac{\gamma_n \xi_n \rho_{n+1}}{\gamma_{n+1} \tau_n \rho_n};$ 
20:     $\tau_{n+1} = \frac{\varepsilon_{n+1}}{\rho_{n+1}} - \gamma_{n+1} \mu_{n+1};$ 
21:     $\theta_n = \frac{\tau_n^2 (1 - \lambda_n)}{\lambda_n \tau_n^2 + \gamma_{n+1}^2};$ 
22:     $\kappa_n = \frac{-\gamma_n \tau_n \kappa_{n-1}}{\lambda_n \tau_n^2 + \gamma_{n+1}^2};$ 
23:     $\lambda_n = \frac{\lambda_{n-1} \tau_{n-1}^2}{\lambda_{n-1} \tau_{n-1}^2 + \gamma_n^2};$ 
24:     $d_n = \theta_n d_{n-1} + \kappa_n p_n;$ 
25:     $f_n = \theta_n f_{n-1} + \kappa_n u_n;$ 
26:     $x_n = x_{n-1} + d_n;$ 
27:     $r_n = r_{n-1} - f_n;$ 
28:  end if
29: end for

```

---

The improved QMR method using Lanczos algorithm as underlying process can be efficiently parallelized as follows:

- The inner products of a single iteration step (12), (15), (16), (17) and (18) are independent.
- The matrix-vector multiplications of a single iteration step (7) and (8) are independent.
- The communications required for the inner products (12), (15), (16), (17) and (18) can be overlapped with the update for  $p_n$  in (11).

Therefore, the overhead of communication on parallel distributed memory computers can be significantly reduced.

## 7.5 THE PERFORMANCE MODEL

Based on these mathematical background described above, we will make the following assumptions suggested in [4, 5, 13] for our performance model. First, the model assumes perfect load balance and the processors are configured as *ring*, *d-dimensional mesh with wrap-around connection*, *hypercube* and *balanced binary tree*, respectively. For *d-dimensional mesh* without wrap-around connection, the basic communication operation increase at most by a factor of four. In this model, Each processor holds a sufficiently large number of successive rows of the matrix, and the corresponding sections of the vectors involved. That is, our problems have a strong data locality. Secondly, we can compute the inner products (reduction) in two steps because the vectors are distributed over the processor topology. The computation of an inner product is executed by all processors simultaneously without communication or locally and these partial results are combined by a global sum operation and accumulated at a single destination processor, called *single node accumulation* (SNA). The second phase consists of reversing the directions and sequence of messages sending the final results from this single processor to all other processors, called *single-node broadcast* (SNB).

With regards to the communication model, the *Store-and-Forward routing* (SF) and *Cut-Through routing* (CT) scheme [11] are discussed in our qualitative performance analysis. In SF routing, when a message is traversing a path with multiple links, each intermediate processor on the path forwards the message to the next processor after it has received and stores the entire message. In CT routing, each intermediate processor on the path does not wait for the entire message to arrive before forwarding the message to the next processor. As soon as a smaller unit called a *flit* is received at an intermediate processor, the flit is passed on to the next one. The corresponding communication cost for SNB and inner products with SF and CT routing can be summarized as follows:

where the  $t_s$  is the start-up time that required to initiate a message transfer at the sending processor. The per-word time  $t_w$  is the time each word takes to traverse two directly-connected processors. The per-hop time  $t_h$  is the time needed by the header of a message to travel between two directly-connected processors. The communication cost for SNB with CT routing on the hypercube we list is not optimal. The optimal one is  $2(t_s \log_2 p + t_w m)$ .

In the following part we will describe a simple performance model including the computation time and communication cost for the main kernels as we presented before based on our assumptions. These two important terms are used in our paper suggested in [5]:

- Communication Cost : The term to indicate all the wall-clock time spent in communication, that is not overlapped with useful computation.
- Communication Time : The term to refer to the wall-clock time of the whole communication.

In the non-overlapped communication, the communication time and the communication cost are the same term.

Table 7.1 Communication Operation Cost with SF and CT Routings

Topologies	Operations	SF Routing	CT Routing
Ring	SNB	$(t_s + t_w m) \lceil \frac{p}{2} \rceil$	$(t_s + t_w m) \log_2 p + t_h(p - 1)$
	Inner	$(t_s + 5t_w)p$	$2(t_s + 5t_w) \log_2 p + 2t_h(p - 1)$
Mesh	SNB	$d(t_s + t_w m) \lceil \frac{p^{1/d}}{2} \rceil$	$(t_s + t_w m) \log_2 p + dt_h(p^{1/d} - 1)$
	Inner	$d(t_s + 5t_w)p^{1/d}$	$2(t_s + 5t_w) \log_2 p + 2dt_h(p^{1/d} - 1)$
Hypercube	SNB	$(t_s + t_w m) \log_2 p$	$(t_s + t_w m) \log_2 p$
	Inner	$2(t_s + 5t_w) \log_2 p$	$2(t_s + 5t_w) \log_2 p$
Tree	SNB		$(t_s + t_w m + t_h(\log_2 p + 1)) \log_2 p$
	Inner		$2(t_s + 5t_w) + t_h(\log_2 p + 1)) \log_2 p$

The IQMR algorithm contains three distinct computational tasks per iteration

- Two simultaneous matrix-vector products,  $A\tilde{v}_n$  and  $A^T \tilde{w}_{n+1}$  whose computation time are given  $2t_{fl}N/P$ .
- Five simultaneous inner products,  $(\tilde{v}_{n+1}, \tilde{v}_{n+1})$ ,  $(\tilde{w}_{n+1}, \tilde{w}_{n+1})$ ,  $(\tilde{w}_{n+1}, \tilde{v}_{n+1})$ ,  $(s_{n+1}, \tilde{v}_{n+1})$  and  $(r_{n-1}, r_{n-1})$  whose computation time are given by  $(2n_z - 1)t_{fl}N/P$ .
- Nine vector updates,  $q_n$ ,  $\tilde{w}_{n+1}$ ,  $u_n$ ,  $\tilde{v}_{n+1}$ ,  $p_n$ ,  $d_n$ ,  $f_n$ ,  $x_n$  and  $r_n$  which are given  $2t_{fl}N/P$ .

where  $N/P$  is the local number of unknown of a processor,  $t_{fl}$  is the average time for a double precision floating point operation and  $n_z$  is the average number of non-zero elements per row of the matrix.

The complete (local) computation time for the IQMR method is given approximately by the following equation:

$$T_{comp}^{IQMR} = (19 + 2n_z) \frac{N}{P} t_{fl} = f(m) \frac{N}{p}, \quad (7.5)$$

The corresponding global accumulation and broadcast time for 5 simultaneous inner product of IQMR method,  $g(m)$ , is given as follows:

## 7.6 THEORETICAL PERFORMANCE ANALYSIS

In this section, we will focus on the theoretical analysis of the parallel performance of IQMR method where the efficiency, speed-up and runtime are expressed as functions



Table 7.2 Communication Cost with SF and CT Routings

	Topologies	SF Routing	CT Routing
$g(m)$	Ring	$(t_s + 5t_w)p$	$2(t_s + 5t_w) \log_2 p + 2t_h(p - 1)$
	Mesh	$d(t_s + 5t_w)p^{1/d}$	$2(t_s + 5t_w) \log_2 p + 2dt_h(p^{1/d} - 1)$
	Hypercube	$2(t_s + t_w) \log_2 p$	$2(t_s + 5t_w) \log_2 p$
	Tree	$2(t_s + 5t_w) \log_2 p$	$2(t_s + 5t_w) + t_h(\log_2 p + 1) \log_2 p$

of the number of processors scaled by the number of processors that gives the minimal runtime for the given problem size.

The total runtime for IQMR is given by the following equation:

$$T_p^{IQMR} = T_{comp}^{IQMR} + T_{comm}^{IQMR} = f(m) \frac{N}{p} + g(m) \quad (7.6)$$

This equation shows that for sufficiently large  $p$  the communication time will dominate the total runtime.

Let  $p_{max}$  denote as the number of processors that minimizes the total runtime  $T_p$  for any  $p$  processors of IQMR method. The percentage of the computation time in the whole runtime  $E_p = T_1/pT_p$  is defined as parallel efficiency. The speed-up  $S_p$  for any  $p$  processors is  $S_p = T_1/T_p$  and  $\alpha$  be the fraction  $\alpha = p/p_{max}$  and  $\omega = d\alpha^{\frac{d+1}{d}}$ . Then the theoretical performance for SF routing can be summarized as follows:

Table 7.3 Theoretical Parallel Performance with SF Routing

Topologies	$p_{max}$	$E_p$	$S_p$	$T_p$
Ring	$\left(\frac{f(m)N}{t_s+5t_w}\right)^{1/2}$	$\frac{1}{1+\alpha^2}$	$\frac{p}{1+\alpha^2}$	$\frac{1+\alpha^2}{\alpha}(t_s + 5t_w)f(m)N)^{1/2}$
Mesh	$\left(\frac{f(m)N}{t_s+5t_w}\right)^{\frac{d}{d+1}}$	$\frac{1}{1+\omega}$	$\frac{p}{1+\omega}$	$\frac{1+\omega}{\alpha}((t_s + 5t_w)^d f(m)N)^{\frac{1}{d+1}}$
Hypercube	$\frac{\ln 2}{2} \frac{f(m)N}{t_s+5t_w}$	$\frac{1}{1+\alpha \ln p}$	$\frac{p}{1+\alpha \ln p}$	$\frac{1+\alpha \ln p}{\alpha} \frac{2(t_s+5t_w)}{\ln 2}$

For CT routing, we can conclude the similar results which can be described in the following theorem:

**Theorem 1** Let  $p_{max}$ , runtime  $T_p$ , speed-up  $S_p$ , and efficiency  $E_p$  be defined as before for any  $p$  processors with SF and CT routings of the parallel IQMR method. Then we

have similar relations described as follows for both SF and CT routing with different network topologies. The runtime  $T_p$ :

$$T_p^{ring} > (T_p^{tree}) > T_p^{mesh} > T_p^{hypercube}, \quad (7.7)$$

the parallel efficiency  $E_p$ :

$$E_p^{ring} < (E_p^{tree}) < E_p^{mesh} < E_p^{hypercube}, \quad (7.8)$$

the parallel speed-up  $S_p$ :

$$S_p^{ring} < (S_p^{tree}) < S_p^{mesh} < S_p^{hypercube}, \quad (7.9)$$

and the corresponding  $p_{max}$ :

$$p_{max}^{ring} < (p_{max}^{tree}) < p_{max}^{mesh} < p_{max}^{hypercube}. \quad (7.10)$$

### 7.6.1 The impacts of reduction by overlapping

In this part, we will use this model to evaluate the impact in parallel performance due to the the communication reductions by overlapping .

Since in the IQMR method, there is one possibility to overlap the computation time with communication time, we assume  $p_{ovl}$  as the number of processors for which all communication can be just overlapped with computation. The value for  $p_{ovl}$  with d-dimensional mesh topology and SF routing follows from

$$dp^{1/d}(t_s + 5t_w) = 2t_{fl} \frac{N}{p}.$$

It is easy to know the theoretical result:

$$p_{ovl} = \left( \frac{2t_{fl}N}{d(t_s + 5t_w)} \right)^{d/d+1}.$$

Based on these theoretical results, we will discuss these three different situations:

- If  $p < p_{ovl}$ , there is no significant communication visible.
- If  $p > p_{ovl}$ , the overlap is no longer complete and the efficiency decreases again because the communication time increases and the computation time in the overlap decreases.
- If  $p = p_{ovl}$ , the communication time can be just overlapped by the computation time. And accordingly we have linear speed-up and 100% parallel efficiency.

It is easy to show in general case  $p_{ovl} < p_{max}$ .

For the general case of  $d$ -dimensional mesh topology and SF routing, we assume that a fraction  $\beta$  of the computations in vector update can be used to overlap communication in inner products, the runtime of one iteration for  $p$  processors and  $N$  unknowns to be

$$\hat{T}_P = (19 + 2n_z - 2\beta) \frac{t_{fl}N}{p} + \max(2\beta \frac{t_{fl}N}{p}, d(t_s + 5t_w)p^{1/d}).$$

From the expression of the runtime we can easily drive the number of processors  $\hat{p}_{max}$  for which  $\hat{T}_p$  is minimal, for which  $\hat{S}_p$  is maximal, is

$$\hat{p}_{max} = (\frac{(19 + 2n_z - 2\beta)t_{fl}N}{t_s + 5t_w})^{d/d+1}.$$

The percentage of the computation time in the whole runtime  $\hat{E}_p$  for  $\hat{p}_{max}$  processors is :

$$\hat{E}_{\hat{p}_{max}} = \frac{19 + 2n_z}{(d + 1)(19 + 2n_z - 2\beta)}.$$

It shows that  $\hat{p}_{max} > p_{ovl}$ , the overlapping factor  $\beta = 1$ . This leads to around  $\frac{d}{d+1}\hat{T}_{\hat{p}_{max}}$  is spent in communication!

The corresponding parallel speed-up  $\hat{S}_p$  for  $\hat{p}_{max}$  processor is maximal, is:

$$\hat{S}_{\hat{p}_{max}} = \frac{(19 + 2n_z)\hat{p}_{max}}{(d + 1)(17 + 2n_z)} \approx \frac{1}{d + 1}\hat{p}_{max}.$$

The runtime for  $\hat{T}_p$  for  $\hat{p}_{max}$  is minimal:

$$\hat{T}_{\hat{p}_{max}} = (d + 1)((t_s + 5t_w)^d(17 + 2n_z)t_{fl}N)^{\frac{1}{d+1}}.$$

With regards to the runtime, efficiency, and speed-up for number of processors  $P$  with overlapped communication, the corresponding results can be stated in the following theorem:

**Theorem 2** *Let the runtime, parallel speed-up and the parallel efficiency with overlapped communication denote as  $\hat{T}_p$ ,  $\hat{S}_p$  and  $\hat{E}_p$  respectively. Let  $\theta$  be the fraction as  $p = \theta\hat{p}_{max}$ ,  $\phi = (\frac{\beta}{19+2n_z-2\beta})^{d/d+1}$  and  $\psi = 19 + 2n_z - 2\beta$ , then the parallel efficiency  $\hat{E}_p$  is given by*

$$\hat{E}_p = \begin{cases} 1, & \theta \leq \phi \\ \frac{\psi + 2\beta}{\psi(1 + d\theta^{d+1/d})}, & \theta > \phi \end{cases}$$

*the parallel speed-up  $\hat{S}_p$  is given by*

$$\hat{S}_p = \begin{cases} P, & \theta \leq \phi \\ \frac{(\psi + 2\beta)p}{\psi(1 + d\theta^{d+1/2})}, & \theta > \phi \end{cases}$$

and the runtime  $\hat{T}_p$  is given by

$$\hat{T}_p = \begin{cases} (\psi + 2\beta) \frac{t_{IN}}{p}, & \theta \leq \phi \\ \psi (1 + d\theta^{d+1/2}) \frac{t_{IN}}{p}, & \theta > \phi \end{cases}$$

There are some interesting remarks can be made as follows:

- The maximum speed-up in case of overlapped communication is reached for  $\hat{p}_{max}$ . In general  $\hat{p}_{max}$  is always smaller than  $p_{max}$ . Furthermore, the speed-up for  $\hat{p}_{max}$  is always better.
- A direct comparison shows that the optimal performance for the IQMR method with overlapping is better than the approach without overlapping.
- With overlapping we can run faster on much less processors, which, of course, gives a large improvement in efficiency. But the scalability of the IQMR method is not improved by overlapping the communication.
- This conclusion can be extended to ring, tree and hypercube topologies.

## 7.7 CONCLUSION

In this paper, we mainly present the qualitative analysis of the parallel performance with *Store-and-Forward routing* and *Cut-Through routing* schemes and topologies such as *ring*, *mesh*, *hypercube* and *balanced binary tree*. Theoretically it is shown that the hypercube topology can give us the best parallel performance with regards to parallel efficiency, speed-up, and runtime, respectively. We also study theoretical aspects of the overlapping effect in the algorithms. Further numerical experiments are needed to evaluate the theoretical studies.

## References

- [1] H. M. Bücker and M. Sauren. A parallel version of the quasi-minimal residual method based on coupled two-term recurrences. In *Proceedings of Workshop on Applied Parallel Computing in Industrial Problems and Optimization (Para96)*. Technical University of Denmark, Lyngby, Denmark, Springer-Verlag, August 1996.
- [2] H. M. Bücker and M. Sauren. A parallel version of the unsymmetric Lanczos algorithm and its application to QMR. Technical Report KFA-ZAM-IB-9606, Central Institute for Applied Mathematics, Research Centre Jülich, Germany, March 1996.
- [3] E. de Sturler. A parallel variant of the GMRES( $m$ ). In *Proceedings of the 13th IMACS World Congress on Computational and Applied Mathematics*. IMACS, Criterion Press, 1991.

- [4] E. de Sturler. Performance model for Krylov subspace methods on mesh-based parallel computers. Technical Report CSCS-TR-94-05, Swiss Scientific Computing Center, La Galleria, CH-6928 Manno, Switzerland, May 1994.
- [5] E. de Sturler and H. A. van der Vorst. Reducing the effect of the global communication in GMRES( $m$ ) and CG on parallel distributed memory computers. Technical Report 832, Mathematical Institute, University of Utrecht, Utrecht, The Netheland, 1994.
- [6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.
- [7] R. W. Freund, M. H. Gutknecht, and N. M. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM Journal on Scientific and Statistical Computing*, 14:137–158, 1993.
- [8] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.
- [9] R. W. Freund and N. M. Nachtigal. An implementation of the QMR method based on coupled two-term recurrences. *SIAM Journal on Scientific and Statistical Computing*, 15(2):313–337, 1994.
- [10] S. K. Kim and A. T. Chronopoulos. An efficient nonsymmetric Lanczos method on parallel vector computers. *Journal of Computational and Applied Mathematics*, pages 357–374, 1992.
- [11] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, 1994.
- [12] C. Lanczos. An iteration method for the solution of the eigenvalues problem of linear differential and integral operators. *Journal of Research of National Bureau of Standards*, 45:255–282, 1950.
- [13] T. Yang. Solving sparse least squares problems on massively parallel distributed memory computers. In Proceedings of *International Conference on Advances in Parallel and Distributed Computing (APDC-97)*, March 1997. Shanghai, P.R.China.
- [14] T. Yang and H. X. Lin. The improved quasi-minimal residual method on massively distributed memory computers. In Proceedings of *The International Conference on High Performance Computing and Networking (HPCN-97)*, April 1997.
- [15] T. Yang and H. X. Lin. Performance analysis of the IQMR method on bulk synchronous parallel architectures. In Proceedings of *The International Workshop on Computational Science and Engineering (IWCSE'97)*, May 1997.



## **II   PARALLEL NUMERICAL APPLICATIONS**





# 8 PARALLEL ALGORITHM FOR 3D ELASTICITY PROBLEMS BASED ON AN OVERLAPPING DD PRECONDITIONER

Krassimir Georgiev

*Central Laboratory for Parallel Processing  
Bulgarian Academy of Sciences  
Acad. G. Bonchev str., Bl. 25-A, 1113 Sofia, Bulgaria*

georgiev@parallel.bas.bg

**Abstract:** In this paper we are mainly concerned with an additive overlapping domain decomposition (DD) algorithm and its implementation on parallel computers with shared and distributed memory including clusters of workstations. The problem under consideration is the numerical solution of 3D elasticity systems but the main ideas, algorithms and codes are applicable to 3D elliptic boundary value problems with discontinuous coefficients. The system of linear equations which has to be solved after a finite element method discretization is symmetric and positive definite and the Preconditioned Conjugate Gradient method with a preconditioner constructed via Domain Decomposition is used. The algorithm is highly parallelizable. The Message Passing Interface (MPI) standard is used on both shared and distributed memory parallel computer platforms. Results from numerical experiments on two symmetric multiprocessor systems and IBM SP parallel computer with distributed memory are discussed.

## 8.1 INTRODUCTION

In recent years there has been a considerable development in Domain Decomposition (DD) methods for numerical solution of partial differential equations discretized using finite differences and finite elements methods. Among the reasons underlying the interest in these techniques we mention the possibilities of determining effective parallel algorithms and implementation on modern high-speed computers mainly parallel computers with shared and distributed memory including clusters of workstations (see [7,11]) and the references given there). On the other hand DD methods are very

convenient in order to handle complex geometries, nonsmooth solutions, differential equations which have different coefficients in different subregions of the domain.

The problem considered here is described by a coupled system of elliptic partial differential equations of second order, with strongly varying coefficients. This system is discretized by brick elements and trilinear basis function. The discretization leads to the problem of solving a system of linear algebraic equations with sparse, symmetric and positive definite coefficient matrix. The Preconditioned Conjugate Gradient (PCG) method is used to find the solution. Our approach consists of applying a preconditioner constructed via an additive overlapping DD method. The new algorithm is highly parallelizable. The Message Passing Interface (MPI) standard both for shared and distributed memory parallel computers is used.

The numerical experiments reported in this paper are obtained on two symmetric multiprocessor systems: SUN Enterprise 3000 (workstation with eight 167 MHz Ultra SPARC processors) and SUN SPARCstation 20 (a workstation with four 99 MHz processors) as well as the parallel computer with distributed memory IBM SP Power 2 (with 32 120 MHz RS/6000 processors). A set of tables and figures illustrate the performed numerical tests on the above mentioned computers. They well demonstrate the parallel properties of the proposed algorithm as well as the robustness of the developed codes.

The remainder of the paper is organized as follows. Section 8.2 is devoted to a description of the problem under consideration. In Section 8.3 we focus on the overlapping domain decomposition approach. The parallel solver and MPI primitives used can be found in Section 8.4. Section 8.5 consists of numerical tests on two models of SUN symmetric multiprocessor systems and on the distributed memory parallel computer IBM SP. The conclusions, requirements and some outlook for the parallel performance of the developed parallel codes are given in the final Section 8.6.

## 8.2 THE PROBLEM

Let  $\mathcal{B}$  be an elastic body occupying a bounded polyhedral domain  $\Omega \in \mathbb{R}^3$ , and let Dirichlet/Neumann boundary conditions on  $\Gamma \equiv \partial\Omega$  be imposed. We denote by  $\mathbf{u} = [u_1, u_2, u_3]^T$  the displacement vector, by  $\mathbf{f} = [f_1, f_2, f_3]^T$  the vector of the body forces, by  $\underline{\sigma} = (\sigma_{ij})(\mathbf{u})$  the stress tensor and by  $\underline{\epsilon}(\mathbf{u}) = (\epsilon_{ij}(\mathbf{u}))$  the strain tensor (the deformation associated with  $\mathbf{u}$ ).

We consider elasticity problems under the assumptions that the *displacements are small* and the *material properties are isotropic*.

The stressed–strained state of the elastic body under consideration can be described by a coupled elliptic system of three partial differential equations and the corresponding boundary conditions as follows [10]

$$\begin{aligned} A^T \underline{\sigma} + \mathbf{f} &= \mathbf{0} & \forall (x_1, x_2, x_3) \in \Omega \\ \mathbf{u} &= \mathbf{u}_D & \forall (x_1, x_2, x_3) \in \Gamma_D \\ \sum_{i=1}^3 \sigma_{ij} n_i &= u_{N_j} & \forall (x_1, x_2, x_3) \in \Gamma_N, \quad j = 1, 2, 3, \end{aligned}$$

where

$$A = \begin{pmatrix} \frac{\partial}{\partial x_1} & 0 & 0 \\ 0 & \frac{\partial}{\partial x_2} & 0 \\ 0 & 0 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} & 0 \\ 0 & \frac{\partial}{\partial x_3} & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} & 0 & \frac{\partial}{\partial x_1} \end{pmatrix}.$$

The variational formulation of the problem to compute the displacements of the above determined elastic body under forces  $\mathbf{f}$  is:

Find:  $\mathbf{u} \in (H_0^1(\Omega))^3$ , such that

$$\begin{aligned} a_\Omega(\mathbf{u}, \mathbf{v}) &\equiv \int_\Omega (\lambda \operatorname{div} \mathbf{u} \operatorname{div} \mathbf{v} + \mu \sum_{i,j=1}^3 \epsilon_{ij}(\mathbf{u}) \epsilon_{ij}(\mathbf{v})) d\mathbf{x} \\ &= F(\mathbf{v}), \quad \forall \mathbf{v} \in (H_0^1(\Omega))^3, \end{aligned}$$

where  $F(\mathbf{v}) = \sum_{i=1}^3 \int_\Omega f_i v_i d\mathbf{x}$ . The positive material coefficients  $\lambda$  and  $\mu$  depend on the modulus of elasticity (Young's modulus) -  $E$  and on the contraction ratio (Poisson ratio) -  $\nu$ .

The bilinear form  $a_\Omega(\mathbf{u}, \mathbf{v})$  is second order, symmetric and coercive.

The corresponding discrete variational problem reads as follows::

Find:  $\mathbf{u}_h \in V^h(\Omega) \subset (H_0^1(\Omega))^3$ , such that

$$a_\Omega(\mathbf{u}_h, \mathbf{v}_h) = F(\mathbf{v}_h), \quad \forall \mathbf{v}_h \in V^h(\Omega).$$

In this study we use the standard trilinear nodal basis functions  $\Phi_i$  for the space  $V^h$  [2]. If we express the approximate solution by the basic functions i.e.

$$u_h^{(k)} = \sum U_i^{(k)} \Phi_i, \quad k = 1, 2, 3$$

we will obtain the linear system of equations

$$K \mathbf{U} = \mathbf{F}, \quad (8.1)$$

where  $\mathbf{U} = [U_i]^T$  is the vector of unknowns —  $U_i$ ,  $i = 1, 2, \dots, N$ ,  $\mathbf{F}$  — the right-hand side is the vector with components  $F_i = \mathbf{F}(\Phi_i)$ , and  $K$  is the so called stiffness matrix with entries  $K_{ij} = a_\Omega(\Phi_i, \Phi_j)$ .

To solve the linear system of equations (1) the preconditioned conjugate gradient procedure is used [1]. In order to obtain the *initial iterate* -  $\mathbf{U}_0$  and the *next search direction* -  $\mathbf{z}_k$  in the iterative procedure, the systems

- $M \mathbf{U}_0 = \mathbf{F}$
- $M \mathbf{z}_k = \mathbf{r}_k$

with the preconditioning matrix  $M$  have to be solved.

### 8.3 HOW THE DOMAIN DECOMPOSITION PRECONDITIONER IS CONSTRUCTED

In 1936, Sobolev [13] showed that the Schwarz alternating algorithm for the two subdomains case converges for the linear elasticity equations. In the developed parallel algorithm we use the Schwarz algorithm based on many overlapping subdomains .

The triangulation of the computational domain  $\Omega$  may be done in the following way. The domain is first divided [8] into non-overlapping substructures  $\Omega_i, i = 1, 2, \dots, p$ . All this substructures are further divided into parallelepipedal elements (bricks) with size  $(h_1 \times h_2 \times h_3)$ . We extend each subdomain  $\Omega_i$  to a larger region  $\hat{\Omega}_i$  with an overlap of  $O(h)$ , where  $h = \max(h_1, h_2, h_3)$ . We also assume that  $\partial\hat{\Omega}_i$  does not cut through any brick element. We make the same construction for the substructures that are next to the boundary of the domain  $\Omega$  except that we cut off the part of  $\hat{\Omega}_i$  that is outside of  $\Omega$ .

Let  $\hat{K}_i$  be a local stiffness matrix according to the subdomain  $\hat{\Omega}_i$ . We define restriction maps  $R_i$  and extension maps  $R_i^T$ , as follows. Let the global stiffness matrix  $K$  is of order  $N = 3 \times n_1 \times n_2 \times n_3$ , where  $n_1, n_2$  and  $n_3$  are the number of the nodes in  $x_1, x_2$  and  $x_3$  directions for the whole computational domain. Let us denote with  $\hat{N}_i$  the number of the unknowns corresponding to the interior with respect to  $\hat{\Omega}_i$ , nodes. For each subdomain  $\hat{\Omega}_i$  the matrix  $R_i$  is  $(N \times \hat{N}_i)$  restriction matrix with entries 1's and 0's, taking into account the indexes of the nodes belong to  $\hat{\Omega}_i$ . This matrix restricts a vector  $\mathbf{U}$  of length  $N$  to  $R_i \mathbf{U}$  of length  $\hat{N}_i$ . Thus, the local stiffness matrix  $\hat{K}_i$  corresponding to the subdomain  $\hat{\Omega}_i$  is

$$\hat{K}_i = R_i K R_i^T.$$

Let us note that in the case of many subdomains, the additive Schwarz algorithm is a generalization of the well known two-subdomain case with a preconditioner

$$M_{add}^{-1} = \sum_{i=1}^p R_i^T \hat{K}_i^{-1} R_i.$$

Indeed, if we start with an iterate  $(k)$ , we compute  $(k+1)$  by

$$\mathbf{U}^{(k+1)} = \mathbf{U}^{(k)} + \left( \sum_{i=1}^p R_i^T \hat{K}_i^{-1} R_i \right) (\mathbf{F} - K \mathbf{U}^{(k)})$$

As  $R_i^T \hat{K}_i^{-1} R_i \mathbf{y}$  can be computed in parallel for the different subdomains, this leads to a coarse grain parallelism. Taking into account the definition of the restriction matrix  $R_i$  (and the extension or interpolation matrix  $R_i^T$ ) we may conclude that it is not necessarily to store these matrices.

So defined preconditioner  $M_{add}$  is a straightforward generalization of the standard block-Jacobi preconditioner. The convergence rate of this preconditioner deteriorates when the number of subdomains increases [7]. There are several techniques which

improve the properties of the above considered algorithm. The price has to be paid is communications. One common mechanism for that is when the fine grid (characteristic diameter  $h$  of the elements) is a refinement of the coarse mesh (the mesh obtained from the subdomains) [6,7]. Then if we let  $R_H^T$  to be the standard interpolation map of coarse grid functions to the fine grid functions (like in the two-level multigrid methods) then its transpose  $R_H$  is a weighted restriction map. If now  $K_H$  is the coarse grid discretization matrix, i.e.  $K_H = R_H K R_H^T$  then the improved additive Schwarz preconditioner can be defined by the preconditioner

$$M_{add, impr}^{-1} = R_H^T K_H^{-1} R_H + \sum_{i=1}^p R_i^T \hat{K}_i^{-1} R_i.$$

In this paper we do not consider the latest algorithm.

The algorithm for solving the system  $M \mathbf{z}_k = \mathbf{r}_k$  with  $M = M_{add}$  is as follows:

- $\hat{\mathbf{r}}_k^i$  — vector of a length  $\hat{N}_i$ ; entries are equal to those of  $\mathbf{r}_k$  for the components corresponding to the mesh points in  $\hat{\Omega}_i$  and 0 elsewhere;
- $\hat{\mathbf{z}}_k^i$  :  $\hat{K}_i \hat{\mathbf{z}}_k^i = \hat{\mathbf{r}}_k^i$ ;
- $\mathbf{z}_k^i$  — vector of a length  $N$ ; an extension of  $\hat{\mathbf{z}}_k^i$  with 0 entries outside  $\hat{\Omega}_i$ ;
- $\mathbf{z}_k = \sum_{i=1}^N \mathbf{z}_k^i$

All four steps of this algorithm can be done in parallel. The coefficient matrices of the systems of linear algebraic equations

$$\hat{K}_i \hat{\mathbf{z}}_k^i = \hat{\mathbf{r}}_k^i$$

are symmetric positive definite block matrices and the preconditioned conjugate gradient method with **Block-Size Reduction Block-Incomplete LU** preconditioner proposed by Chan and Vassilevski [4] (see also: [3] and [5]) is used to solve the systems in each subdomain.

## 8.4 THE PARALLEL SOLVER

When parallel computers (with shared or distributed memory, and including here the clusters of workstations) are used for an implementation of domain decomposition algorithms, as a rule the different subdomains are mapped to individual processors.

The problem which has to be solved is the communications between the different processors. The standard Message Passing Interface (MPI) [9, 12] is used.

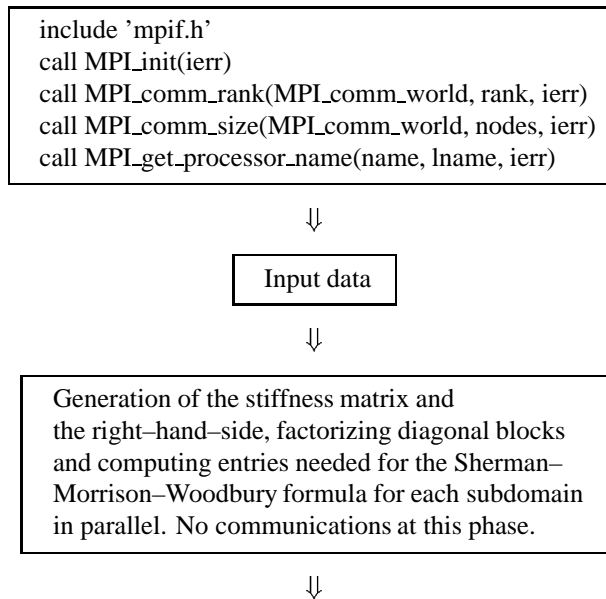
At the recent release of the codes realizing our algorithm for use on parallel computers we have used the following MPI primitives:

- **blocking send** (MPI\_SEND) and **blocking and unblocking receive** (MPI\_RECV and MPI\_IRecv) operations in a **standard mode** from the *Point-to-Point communications*;

- MPI\_BARRIER for **barrier synchronization**, MPI\_BCAST for **broadcasting** data, MPI\_GATHER and MPI\_GATHERV (a vector variant of MPI\_GATHER) for **gathering** data, MPI\_ALLREDUCE for **reduction** operations from the *Collective communications*;

- MPI\_WTIME for **timing** (*elapsed time*) the programs or sections of them.

An outline of the code is following.



**Solving the system (1) via PCGM  
with DD preconditioner**

**Computing the initial iterate in parallel**

PCGM with BSR-MILU preconditioner.  
Each processor consists of its own part of  
the vector of the initial iterate.  
No communications at this phase.

\*\*\*\*\*

**Computing the initial defect in parallel.**

This phase consist of **matrix–vector product** and  
**addition of vectors** and therefore each  
processor has to send/receive a part of the vectors  
corresponds to the overlapping regions to/from  
his neighbors.

After that the processors work in parallel  
and each processor consists of its own part  
of the vector of the initial defect.

*MPI communication primitives used:*

MPI\_SEND, MPI\_RECV, MPI\_WAITALL

\*\*\*\*\*

**Computing the initial quasidefect in parallel**

PCGM with BSR-MILU preconditioner.  
Each processor consists of its own part of the vector  
of the initial quasidefect.  
No communications at this phase.

\*\*\*\*\*

**Iteration cycle:**

Dot and matrix–vector products, vectors additions.

*MPI communication primitives used:*

MPI\_SEND, MPI\_RECV, MPI\_WAITALL,  
MPI\_ALLREDUCE, MPI\_SUM.

The computing of the next search direction, which  
leads to solving systems of equations,  
is done in parallel in each subdomain without  
any communications  
(PCGM with BSR-MILU preconditioner).



Output data



call MPI\_FINALIZE(ierr)

## 8.5 NUMERICAL EXPERIMENTS

### 8.5.1 The test problem

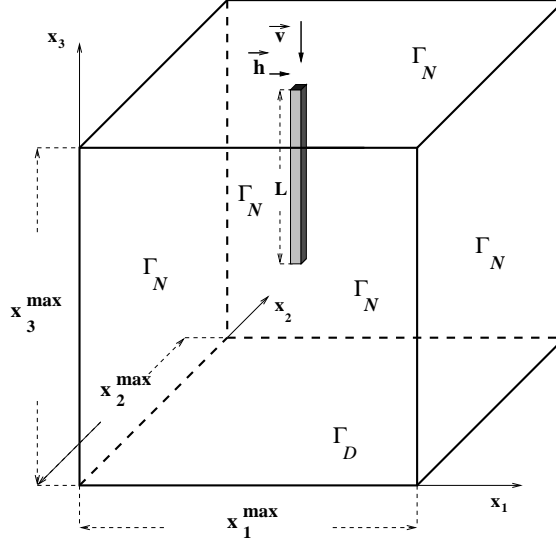


Figure 8.1 The computational domain

Consider a single pile in a homogeneous sandy clay soil layer. The computational domain is the parallelepiped  $\Omega = [0, x_1^{max}] \times [0, x_2^{max}] \times [0, x_3^{max}]$  (Fig.1).

Boundary conditions of Dirichlet and Neumann type on the different parts of the boundary of the domain are as follows:

- Homogeneous Dirichlet boundary conditions on the *bottom side* of the parallelepiped  $x_3 = 0$ , i.e the displacements are equal to zero;
- Homogeneous Neumann boundary conditions on the *vertical sides* and on the *top side* of the parallelepiped

$$x_1 = 0, \quad x_1 = x_1^{max}, \quad x_2 = 0, \quad x_2 = x_2^{max}, \quad x_3 = x_3^{max}$$

i.e the stresses are equal to zero, excluding the pile surface, where the load from the upper construction is applied;

The computational domain  $\Omega$  is determined by

$$x_{max} = y_{max} = 12m; \quad z_{max} = 27m.$$



The length of the pile and the size of the active zone under the top of the pile are respectively  $L = 15m$  and  $H_{act} = 12m$ . The mechanical characteristics of the pile and of the soil layer are:

- (a) *pile*:  $E = 31500 MPa$ ;  $\nu = 0.2$ ;  
 (b) *soil*:  $E = 10 MPa$ ;  $\nu = 0.3$ .

An uniformly load distribution on the cross section of the pile is assumed.

### 8.5.2 Partitioning of the domain

We divide the computational domain in strips according to  $x_3$ -coordinate direction and depending on the number of the processors of the parallel computing system we are going to use. The case  $p = 4$  and  $n_1 = n_2 = n_3 = 32$  with an overlap of  $2h_3$  is presented on Fig.2.

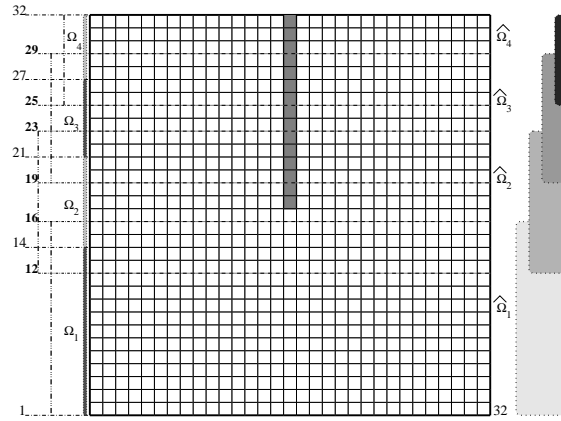


Figure 8.2 The four subdomain case. Cross section  $x_2 = \frac{x_2^{max}}{2}$ . The overlap is  $2h_3$ .

When we perform an algorithm on parallel computers we have to pay attention that all the processors or many of them work in parallel. It is a difficult task to keep all processors busy all of the time. Often, some processors have to wait for others to finish their tasks. It is well seen that even when only one processor have more work to do then others, for example,  $p - 1$  processors have to do work measured with  $W_1$  and the  $p$ -th processor has to do work -  $W_2$ , where  $W_2 > W_1$ , the best speedup which is able to be found is

$$S_p = (p - 1) W_1 + \frac{W_1}{W_2} = 1 + (p - 1) \frac{W_1}{W_2}.$$

Thus, the processors which work in parallel must coordinate their work. Note, that in almost all algorithms in practice, the difference between  $W_1$  and  $W_2$  is not a single arithmetic operation but one or more subroutines. Unfortunately, there is no an efficient, perfectly parallel algorithm for solving any partial differential equations (PDE's) numerically. In 1988, Worley [14] showed that perfectly parallel algorithms for partial differential equations do **not exist** and that for a prior given accuracy there is a **lower bound** of the time it will take to achieve this accuracy.

In order to avoid some *load imbalances* we do not divide the whole computational domain on equal in  $x_3$ -direction subdomains. The main reason for the load imbalances in the proposed algorithm is that we have to solve a coupled system of elliptic partial differential equations, with strongly varying coefficients, and we use for that purpose an iterative procedure which rate of convergence, i.e. number of iterations, depend on the ratio between the coefficients.

### 8.5.3 Results from the numerical experiments

Numerical tests of the code realizing the overlapping domain decomposition approach for the 3D simulation of pile system were done on the symmetric multiprocessor computers SUN Enterprise 3000 (up to eight processors) and SUN SPARC 20 (up to four processors) as well as on the parallel computer with distributed memory IBM SP Power 2 (up to eight processors). The *total elapsed time* ( $T_p$ ) for executing the code on  $p$  processors measured by MPI routine MPI\_WTIME) in seconds, the *speedup* ( $S_p = T_1/T_p$ ) and the *parallel efficiency* ( $E_p = S_p/p$ ) obtained during the numerical experiments can be found in the next three tables.

Table 8.1 Elapsed time, speedup and parallel efficiency for SUN SPARC station 20

Proc.	$T_p$	$S_p$	$E_p$
1	4354	-	-
2	2816	1.07	0.54
4	832	3.63	0.91

As it was expected, the computational time decreases when the number of the processors (subdomains) increases with a factor proportional to the ratio between processors used in the different runs. In the case when the computational domain is divided only in two overlapping subdomains the elapsed time decreases too little and does not scale. The reason is that when the domain is divided in two overlapped subdomains, the subdomain where the pile is located is quite large and the number of the inner iterations (PCG-BSR-BILU) are only approximately two times less than in the algorithm without subdividing. the fact that some domain decomposition iterations

Table 8.2 Elapsed time, speedup and parallel efficiency for SUN Enterprise 3000

Proc.	$T_p$	$S_p$	$E_p$
1	1108	-	-
2	1032	1.07	0.54
4	301	3.68	0.92
8	153	7.24	0.90

Table 8.3 Elapsed time, speedup and parallel efficiency for IBM SP Power 2

Proc.	$T_p$	$S_p$	$E_p$
1	619	-	-
2	389	1.59	0.80
4	160	3.87	0.97
8	74	8.36	1.05

(outer iterations) have to be performed explains the worse results for this case (row 1 in both tables).

The superlinear speedup which can be seen in some of the runs can be explained with *the increased memory locality, the cash locality and the communication overhead*.

The developed code is a portable code for parallel computers with distributed memory, symmetric multiprocessor computers and clusters of workstations with MPI installation. All the runs on the different computer systems were done with the same code but with the particular compiler. The code is not optimized for the particular computer used which is very important in order to exploit fully the great potential ability of the different computers.

## 8.6 CONCLUSIONS AND OUTLOOK

Our numerical tests for the codes realizing the overlapping domain decomposition technique for solving the problem arising from the 3D simulation of pile system show that this approach is an effective tool to determine high-performance parallel algorithms for implementation on symmetric multiprocessor and parallel computers with distributed memory up to eight processors. As the clusters of workstations can be

modeled as distributed memory computers with the only difference that the network parameters (the start-up time and the time for transferring of single word) are quite larger, one can expect than the same conclusion would be true for this class of parallel computer systems. In fact, there are two main possibilities to use workstations for implementation of the overlapped domain decomposition algorithms: to use a single workstation and exploit a virtual memory or to use a network of workstations. Why not use a single workstation? The answer is that the page thrashing reduces the effective computation rate.

The partitioning of the computational domain in several subdomains allows to solve this large-scale problem because of possibility to store different part of the matrices into the memory of the different computers, when clusters of workstations are used or into the memory of different processors, where distributed memory parallel computers are used, and solve efficiently huge real-life problems, when the problem does not fit in the memory of one processor. Moreover, the computational work may be done in parallel for the big part of the algorithm. The use of the standard message passing interface (MPI) is a key component in the development of concurrent computing environment in which applications and tools can be transparently ported between different computers.

This particular implementation of the domain decomposition approach for 3D simulation of pile system is a portable code for parallel computers with distributed memory, symmetric multiprocessor computers and clusters of workstations. In order to improve the results one has to optimize the implementation and tune it for every particular computer platforms using some special requirements for optimal use of each of them.

## Acknowledgments

This research was partly supported by the Ministry of Science and Education of Bulgaria (Grant I-811/98)

## References

- [1] O. Axelsson, *Iterative solution methods*, Cambridge Univ. Press (1994).
- [2] O. Axelsson, V. A. Barker, *Finite element solution of boundary value problems: theory and applications*, Orlando: Academic Press (1983).
- [3] T. Chan, S. Margenov, P. Vassilevski, Performance of block-ILU factorization preconditioners based on block-size reduction for 2D elasticity systems, *SIAM J. Sci. Comput.*, **18** (1997), 1355-1366
- [4] T. Chan, P. S. Vassilevski, *A framework for block-ILU factorizations using block-size reduction*, *Math. Comp.* 64 (1995), 129-156
- [5] T. Chan, P. S. Vassilevski, *Convergence analysis of block-ILU factorization algorithms based on block-size reduction*, CAM Report, **95-46** (1995), Math. Dept., UCLA
- [6] M. Dryja, B. Smith, O. Widlund, *Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions*, Tech. Report 615, Dept. of Computer Science, Courant Institute, New York, NY (1993)

- [7] M. Dryja, O. Widlund, *Additive Schwarz methods for elliptic finite element problems in three dimensions*, in: Fifth Conf. on Domain Decomposition Methods for Partial Differential Equations, T. Chan, D. Keyes, G. Meurant, J. Scroggs, R. Voigt, Eds., SIAM, Philadelphia, PA, (1991)
- [8] K. Georgiev, *Implementation of the additive overlapping domain decomposition method to 3-D elasticity problems*, *Mathematica Balkanica*, **10** (1997), 419-433
- [9] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts (1996)
- [10] J. L. Meek, *Computer methods in structural analysis*, Fong @ Sons Printers (1991)
- [11] B. Smith et al, *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press (1996)
- [12] M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongara, *MPI: The Complete Reference* The MIT Press, Cambridge, Massachusetts (1996)
- [13] S. Sobolev, *L'Algorithme de Schwarz dans la Théorie de l'Elasticité*, *Comptes Rendus (Doklady) de l'Académie des Sciences de l'URSS*, **IV** (1936), 1212-1220
- [14] P. Worlay, *Information requirements and the implications for parallel computation*, Tech. Report STAN-CS-88-1212, Dept. of Computer Science, Stanford University (1988)



# **9     DISTRIBUTED CONTROL PARALLELISM FOR MULTIDISCIPLINARY DESIGN OF A HIGH SPEED CIVIL TRANSPORT**

Denitza T. Krasteva,  
Layne T. Watson, Chuck A. Baker, Bernard Grossman, William H. Mason,

*Multidisciplinary Analysis and Design (MAD) Center for Advanced Vehicles,  
Virginia Polytechnic Institute and State University,  
Blacksburg, Virginia 24061-0106, U.S.A.*

ltw@cayuga.cs.vt.edu

Raphael T. Haftka

*Department of Aerospace Engineering, Mechanics and Engineering Science,  
University of Florida, Gainesville, Florida 32611-6250, U.S.A.*

**Abstract:** Large scale multidisciplinary design optimization (MDO) problems often involve massive computation over vast data sets. Regardless of the MDO problem solving methodology, advanced computing technologies and architectures are indispensable. The data parallelism inherent in some engineering problems makes massively parallel architectures a natural choice, but efficiently harnessing the power of massive parallelism requires sophisticated algorithms and techniques. This paper presents an effort to apply massively scalable distributed control and dynamic load balancing techniques to the reasonable design space identification phase of a variable complexity approach to the multidisciplinary design optimization of a high speed civil transport (HSCT). The scalability and performance of two dynamic load balancing techniques, random polling and global round robin with message combining, and two termination detection schemes, token passing and global task count, are studied. The extent to which such techniques are applicable to other MDO paradigms, and to the potential for parallel multidisciplinary design with current large-scale disciplinary codes, is of particular interest.

## Introduction

The requirement for timely deliverance, in the context of inherent computational complexity and huge problem size spanning several disciplines, is typical of modern large-scale engineering problems (e.g., aircraft design). This has provided the driving force for research in the area of multidisciplinary design optimization (MDO) to develop practical, scalable methodologies for design optimization and analysis from the perspective of more than one discipline. The computational intensity of realistic multidisciplinary design optimization problems presents a major obstacle and bottleneck. For this reason, high performance computing and its efficient use constitute a very important MDO tool. There is an ongoing effort amongst engineering and scientific computing researchers to build sophisticated parallel and distributed algorithms for the solutions of specific types of problems, such as computational fluid dynamics, partial differential equations, finite element analysis, etc. (see, for example, the July 1998 issue of *Advances in Engineering Software*). Despite their good performance and promising potential, such codes are not widely integrated in MDO environments, since it is a nontrivial task to efficiently blend heterogeneous, disciplinary engineering codes together. Obstacles that arise are complex interactions between disciplines, incompatible interfaces, nonstandard programming practices, lack of detailed documentation, and sometimes failure to scale up to the sizes of realistic MDO problems. As a result, more customization is needed than is feasible. Burgee et al. [7] discuss similar difficulties in their effort to parallelize legacy sequential MDO codes.

Several efforts are described in the literature that propose parallel and distributed solutions to the complexity and computational burden of large scale MDO problems. One strain of research develops methodologies for MDO problem modeling and formulation with the goal of creating significant opportunities for distributed and parallel computation. Kroo et al. [23] propose two such methodologies. One is the decomposition of analyses into simpler modules with limited interdependencies, so that each module can be run concurrently. Collaborative optimization [5], on the other hand, aims at modeling the entire design process as a collaboration between parallel tasks/disciplines, under the auspices of a centralized coordinating process. Dennis and



Lewis introduce the “individual discipline feasible” [8] problem formulation approach for MDO that has the advantage of using third party disciplinary analysis codes. There is work at Georgia Tech on agent based technologies for the IMAGE infrastructure of their decision support integrated product and process development (IPPD) architecture DREAMS [15]. The applicability and scalability of the above methods for large-scale systems has yet to be established.

A second strain of research is oriented towards the design and implementation of generic MDO computing frameworks that support concurrent execution across distributed, heterogeneous platforms (see Access Manager [25], COMETBOARDS [18], and Fido [32]). For example, Wujek et al. [33] propose a framework, later extended by Yoder and Brockman [34], that facilitates distributed collaborative design [23] and manages the entire problem-solving life-cycle through a graphical user interface. These systems aim to automate the design optimization process by controlling computing processes, and tracking and monitoring intermediate results. Their problem definition capabilities need to be very flexible and robust in order to accommodate complex MDO problems. Additionally, tracking and monitoring tools like FIDO’s Spy [32] are very important to keep the designer informed about the state of the optimization, and to allow the designer to interact with the automated process. Currently, more empirical evidence is needed to show if automated, “push-button” (see Dennis and Lewis [8]) systems are well suited to large, complex MDO problems.

Finally, there has been research on using sophisticated parallel algorithms for different MDO subproblems such as optimization, analysis, etc. For example, Burgee et al. [7] and Balabanov et al. [1] implemented coarse-grained parallel versions of existing analysis and optimization codes for a High Speed Civil Transport. The results were reasonable, but speedup tapered off for less than 100 nodes, due to I/O overhead, and other factors discussed in [7], [1]. Their conclusion was that fine-grained parallelism and reduced I/O versions of the codes would improve scalability. There are some reports of efficiency achieved on massively parallel architectures, i.e., scalability to thousands of nodes. For example, Dennis and Torczon [9] developed the parallel direct search methods for derivative free optimization that are blatantly parallel. Direct search methods are suitable for problems with a relatively small number of design variables ( $< 50$ ). Ghattas and Orozco have developed a parallel reduced Hessian sequential quadratic programming (SQP) method for shape optimization [12] that scales very well up to thousands (16000) of processors for relatively small numbers of design variables ( $< 50$ ), but performance quickly degrades as the number of design variables increases. Eldred et al. [11] have developed an object oriented software framework for generic optimization that experienced optimal performance at 512 processors for a certain test problem, but started degrading after that. For more research on parallel and distributed MDO tools, including parallel genetic algorithms, and Java based solutions, see [19], [3], [10], [2].

This paper focusses on the effective use of massive parallelism and scalable distributed control applied to the reasonable design space identification paradigm embedded within the problem of the multidisciplinary configuration optimization of a High Speed Civil Transport (HSCT). The approach here uses a variable complexity paradigm [7] where computationally cheap low fidelity techniques are used together

with computationally expensive high fidelity techniques throughout the optimization process. Geometric constraints and low fidelity analysis are applied to define promising regions in the design space and to identify intervening/important variables/terms for surrogate models. Higher fidelity analyses are used to generate smooth response surfaces for those regions, which are then analyzed by the optimizers in search of local optima. Typical configuration designs are comprised of 5 to 50 design variables.

The paradigm of reasonable design space identification consists of performing millions of low fidelity analyses at extreme points in a box around a nominal configuration. A single design evaluation takes a fraction of a second on a slow processor, but as the number of design variables grows, millions of evaluations require a significant amount of time. Such an evaluation is too fine grained to lend itself to task parallelism, and so is taken as the atomic grain of computation. In terms of data parallelism, the problem is irregular because each configuration that fails preliminary analysis (violates feasibility constraints) has to be moved towards the center of the box until it is coerced to a reasonable design. This results in a variable number of analyses and time per configuration. Initially, a parallel implementation was developed where all configurations to be evaluated were spread evenly across available processors. A severe load imbalance, where total idle time amounted to one half of total processing time, was observed. Thus dynamic load balancing strategies, so that the load can be effectively redistributed amongst processors at run time, are essential. Two receiver initiated distributed load balancing algorithms—random polling (RP) and global round robin with message combining (GRR-MC)—were implemented.

Load balancing causes remapping of jobs to processors so that processors that have finished their work at some point in time can resume work with a new load. Thus a processor having no load at a certain point in time does not signify that there is no more work to be done. A termination detection algorithm is needed to assert global termination of the system. Two complementary termination detection schemes—global task count and token passing—have been implemented. Section 9.1 describes the reasonable design space paradigm, Section 9.2 the algorithms, and Section 9.3 their implementation. Section 9.4 discusses the results in terms of scalability and efficiency, and Section 9.5 offers some conclusions and possible future work.

## 9.1 HSCT CONFIGURATION OPTIMIZATION

The parallelization techniques described in the following sections are applied in the context of designing an optimal supersonic aircraft with a capacity for 251 passengers, minimum range of 5,500 nautical miles, cruise speed of Mach 2.4, and ceiling height of 70,000 ft. The problem is formulated as a constrained optimization.

$$\min_{x_{min} \leq x \leq x_{max}} f(x), \quad \text{subject to } g_i(x) \leq 0 \quad \text{for all } i \in \{1, \dots, m\},$$

where  $f : \mathbf{R}^n \rightarrow \mathbf{R}$  is the objective function,  $x \in \mathbf{R}^n$  is a vector of  $n$  design variables, and  $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$  is a vector of  $m$  constraints. The values of the design variables in vector  $x$  are lower and upper bounded by  $x_{min}$  and  $x_{max}$  respectively.

Takeoff gross weight (TOGW), expressed as the aggregate of payload, fuel, structural and nonstructural weights, serves as the selected objective function. TOGW is dependent on many of the engineering disciplines involved, (e.g., structural design determines empty aircraft weight, aerodynamic design affects required fuel weight, etc.), and thus provides a measure of merit for the HSCT as a whole. Giunta [13] suggests that minimized takeoff gross weight is also in some sense related to minimized acquisition and recurring costs for the aircraft. Figure 9.1 [22] illustrates a typical aircraft configuration.

The suite of optimization and analysis tools employed for this problem comprises codes developed by engineers in-house (e.g., vortex lattice subsonic aerodynamics, panel code for supersonic aerodynamics) and by third parties (e.g., optimizer, weights and structures, Harris [16] wave drag code for supersonic aerodynamics). The analysis tools are of varying complexity and computational expense, and some have coarse-grained parallel implementations. Interactions and coordination amongst programs in the suite are mostly carried out via file I/O.

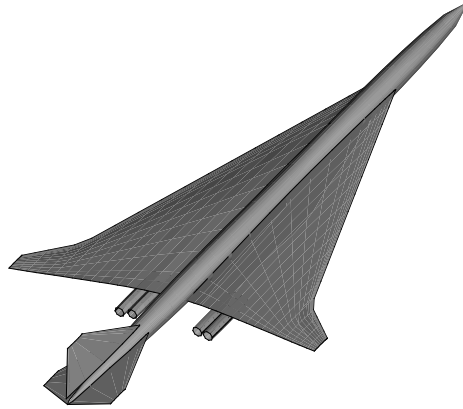
### 9.1.1 Design Variables

Successful aircraft design optimization requires a suitable mathematical characterization of configuration parameters. Typically a configuration has  $n \leq 50$  design variables. In this particular case, 29 variables are used to define the HSCT in terms of geometric—wing-body-nacelle—layout (twenty six variables) and mission profile (three variables). See Table 9.1 [13] for descriptions and typical values of all design variables. The wing is parametrized with eight variables for planform (see Figure 9.2 [13]) and five variables for leading edge and airfoil shape properties (see Figure 9.3 [13]). Two variables express the engine nacelle locations along the wing semi-span. The fuselage shape is defined with eight variables specifying the axial location and radius for each of four restraint points along its fixed 300 ft length. The horizontal and vertical tails are trapezoidal planforms whose areas each comprise a design variable. The thrust of the engine is also a variable. Internal volume of the aircraft is fixed at 23,270 ft<sup>3</sup>.

The idealized mission profile is divided into three segments—takeoff, supersonic leg at Mach 2.4, and landing. There are three variables related to the mission—flight fuel weight, climb rate, and initial supersonic cruise/climb altitude.

### 9.1.2 Constraints

The HSCT optimization process is subject to 69 explicit nonlinear constraints of varying complexity and computational expense. The least expensive to evaluate are geometric constraints that are used to eliminate physically senseless designs involving negative lengths, zero thickness, etc. Aerodynamic and performance constraints vary from moderately expensive (e.g., stability issues) to computationally intensive (range  $\geq 5,500$ ). Table 9.2 [13] lists all constraints with short descriptions.



*Figure 9.1* Typical HSCT configuration.

### 9.1.3 Multi-fidelity Analysis

Minimizing TOGW requires a large number of disciplinary analyses (e.g., structural, aerodynamic), so that the optimal configuration(s) can be found. The computational cost of sophisticated analysis techniques becomes prohibitive as the number of design variables grows ( $\geq 5$ ), and simpler methods are not accurate enough. A multi-fidelity approach employs methods of varying complexity and computational cost, so that optimization becomes feasible.

One method of incorporating the multi-fidelity approach early into the design process is the use of response surface methodologies (RSM). RSM uses simple mathematical models, typically low-order polynomials, to approximate the response and smooth out numerical noise present in the higher fidelity analyses. Such RS models are created using a limited number of analyses at a set of statistically selected points in the design space. The optimizer works with these models (response surfaces) instead of the actual high fidelity analyses, because in addition to smoothing out the noise, once generated the former are much faster and simpler to work with. Additionally, more than one response surface can be generated concurrently.

Unfortunately, the complexity and accuracy of polynomial approximations are adversely affected as the number of design variables increases ( $\geq 20$ ). For this reason, low fidelity analyses are used to reduce the dimensionality and cost of polynomial models by identifying intervening variables and important terms to be used in reduced term models. In addition, close attention must be paid to how the initial data set used to create the RS model is generated.

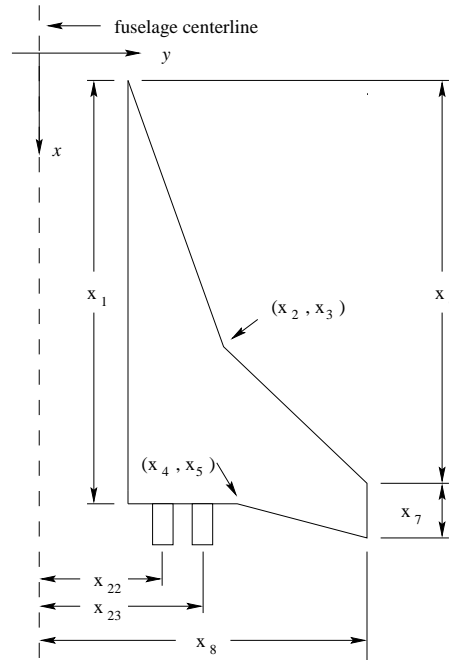


Figure 9.2 Wing planform design variables.

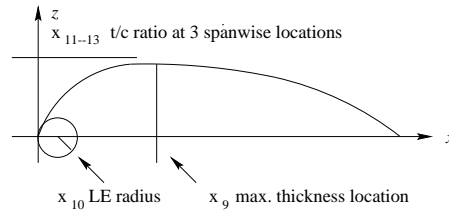


Figure 9.3 Wing airfoil thickness.

## 9.2 PROBLEM DEFINITION

### 9.2.1 Design of Experiments Theory

A point selection algorithm is needed to generate the configurations from the box—a  $p$ -cube, centered at the origin, where  $p$  is the number of design variables—that will be used to define the reasonable design space. A full factorial design is a possible choice, but it will result in an unwieldy number of configurations. For example, a 25 variable problem with two levels for each design variable results in  $2^{25} \approx 33$  million points. With an average of three evaluations needed to bring a point to the reasonable design space, this would require about 100 million low-fidelity analyses. Presently, this

Table 9.1 Design variables and typical values.

<i>Index</i>	<i>Typical</i>	<i>Description</i>
1	181.48	Wing root chord
2	155.90	LE break point, $x$ (ft)
3	49.20	LE break point, $y$ (ft)
4	181.60	TE break point, $x$ (ft)
5	64.20	TE break point, $y$ (ft)
6	169.50	LE wing tip, $x$ (ft)
7	7.00	Wing tip chord (ft)
8	74.90	Wing semi-span (ft)
9	0.40	Chordwise location of max. thickness
10	3.69	LE radius parameter
11	2.58	Airfoil $t/c$ ratio at root, (%)
12	2.16	Airfoil $t/c$ ratio at LE break, (%)
13	1.80	Airfoil $t/c$ ratio at LE tip, (%)
14	2.20	Fuselage restraint 1, $x$ (ft)
15	1.06	Fuselage restraint 1, $y$ (ft)
16	12.20	Fuselage restraint 2, $x$ (ft)
17	3.50	Fuselage restraint 2, $y$ (ft)
18	132.46	Fuselage restraint 3, $x$ (ft)
19	5.34	Fuselage restraint 3, $y$ (ft)
20	248.67	Fuselage restraint 4, $x$ (ft)
21	4.67	Fuselage restraint 4, $y$ (ft)
22	26.23	Nacelle 1 location (ft)
23	32.39	Nacelle 2 location (ft)
24	697.90	Vertical tail area (ft <sup>2</sup> )
25	713.00	Horizontal tail area (ft <sup>2</sup> )
26	39000.00	Thrust per engine (lb)
27	322617.00	Flight fuel (lb)
28	64794.00	Starting cruise/climb altitude (ft)
29	33.90	Supersonic cruise/climb rate (ft/min)

computation is prohibitive, and a scheme based on the partially balanced incomplete block (PBIB) design [17] is used to generate points.

A PBIB of *order*  $n$  consists of points where combinations of  $n$ , usually between two and four, variables at a time change their level. The *level*  $l$  signifies how many different discrete values a variable can assume [4]. For example, a variable allowed to take values in  $\{-1, 0, 1\}$  has level three [21]. The total number of configurations generated by this scheme is

$$\sum_{i=0}^n l^i \binom{p}{i},$$

Table 9.2 Optimization constraints.

<i>Index</i>	<i>Constraint</i>
<b>Geometric Constraints</b>	
1	Fuel volume $\leq$ 50% wing volume
2	Airfoil section spacing at $C_{tip} \geq 3.0$ ft
3–20	Wing chord $\geq 7.0$ ft
21	LE break within wing semi-span
22	TE break within wing semi-span
23	Root chord $t/c$ ratio $\geq 1.5\%$
24	LE break chord $t/c$ ratio $\geq 1.5\%$
25	Tip chord $t/c$ ratio $\geq 1.5\%$
26–30	Fuselage restraints
31	Nacelle 1 outboard of fuselage
32	Nacelle 1 inboard of nacelle 2
33	Nacelle 2 inboard of semi-span
<b>Aerodynamic/Performance Constraints</b>	
34	Range $\geq 5500$ nautical miles
35	$C_L$ at landing speed $\leq 1$
36–53	Section $C_L$ at landing $\leq 2$
54	Landing angle of attack $\leq 12^\circ$
55–58	Engine scrape at landing
59	Wing tip scrape at landing
60	LE break scrape at landing
61	Rudder deflection $\leq 22.5^\circ$
62	Bank angle at landing $\leq 5^\circ$
63	Tail deflection at approach $\leq 22.5^\circ$
64	Takeoff rotation to occur $< V_{min}$
65	Engine-out limit with vertical tail
66	Balanced field length $\leq 11000$ ft
67–69	Mission segments: thrust available $\geq$ thrust required

where  $n$  is the order,  $p$  is the number of design variables, and  $l$  is the level. In effect, this results in all combinations where 1 through  $n$  variables can each take any one of  $l$  values, while the remaining variables are held at a nominal value. The nominal point, where all variables are at their nominal value, is also included ( $i = 0$  in the above formula). For example, the number of configurations that would have to be evaluated when  $l = 2$ ,  $p = 25$ , and  $n = 4$  is

$$\sum_{i=0}^4 2^i \binom{25}{i} = 222051.$$

Clearly this is a much smaller set of points than produced with a full factorial design. Unfortunately, using a PBIB point sample reduces the coverage of the reasonable design space, and optimization can often lead to unexplored corners of the design space.

### 9.2.2 Reasonable Design Space

Generation of the initial design space, using the PBIB design, is followed by the reduction of its volume down to the size of the reasonable design space, containing reasonable HSCT configurations, to exclude from considering unrealistic designs. A reasonable design is a configuration having similar characteristics to the feasible designs, but allowed to have some aerodynamic/performance constraints violated. The elimination of unreasonable designs was desirable, since this would lead to removing most of the designs in the already very sparsely covered design space. Instead, each unreasonable design  $\mathbf{x}$  was moved linearly towards the baseline design  $\mathbf{x}_c$  until it met a certain set of criteria. The set of parameters, used to determine whether a design is reasonable or not, includes 34 standard geometric constraints for HSCT, several additional geometric constraints and some aerodynamic constraints which use "approximate" methods for design analysis and are therefore inexpensive to evaluate. This procedure brings an unreasonable design  $\mathbf{x}$  to the border of the reasonable design space. The design variables were changed linearly with a parameter  $\beta$  between the initial unreasonable design  $\mathbf{x}$  and the central design  $\mathbf{x}_c$ :

$$\mathbf{x}' = \beta (\mathbf{x} - \mathbf{x}_c) + \mathbf{x}_c .$$

The parameter  $\beta$ ,  $0 \leq \beta \leq 1$ , was found for each design using bisection with tolerance 0.01. The parallelization of this paradigm is the focus of this study.

## 9.3 DESCRIPTION OF ALGORITHMS

### 9.3.1 Assumptions

Let  $W$  be the total number of configurations to be evaluated, and let  $N$  be the number of processors available for computation. The following conditions are assumed for the communications network:

- communication channels are reliable, i.e., there is no message loss, corruption, duplication, or spoofing;
- communication channels do not necessarily obey the FIFO rule;
- messages take an unpredictable, but *finite* amount of time to reach their destination;
- a message that has been sent is considered in transition until it has been processed at its destination;



- each processor has knowledge of its own identity;
- the processors are completely connected, i.e., there is either a direct or an indirect communication route from every processor to every other;
- the network is fixed, i.e., its does not change size dynamically. Thus each processor has knowledge of the total number of processors in the network.

It is a prerequisite for the dynamic load balancing and termination detection paradigms described below that initially all work is distributed evenly amongst all processors. Thus, every node starts off with an initial load equal to approximately total work divided by number of processors  $W/N$ . For the remaining part of this section *node* and *processor* will be used interchangeably, and *task*, *work*, and *load* shall refer to the process of evaluating and possibly coercing a configuration, represented by a row of the PBIB design matrix, to a reasonable design.

### 9.3.2 Dynamic Load Balancing

Both algorithms described in this section have the following attributes.

- **Nonpreemptive:** partially executed tasks are not transferred. Preemption can be very expensive in terms of transferring a task's state.
- **Receiver initiated:** work transfer is initiated by receiving nodes. This is more suitable here, since total work is fixed, and there is no good heuristic for estimating if a node is comparatively overloaded, i.e., how long a task will take.
- **Threshold transfer policy:** a node starts looking for more work when the number of its tasks has dropped below a certain threshold.
- **Fixed ratio splitting policy:** when a node is about to transfer work, it uses a fixed ratio  $\alpha$  to split its work  $W_i$  to send away  $\alpha W_i$ .  $\alpha$  is fixed because the algorithms will not be collecting any system information to help them adapt  $\alpha$  to the global state. ( $\alpha = 0.5$  for the results here.)
- **No information policy:** the nodes do not attempt to gather any information about the system state. The potential overhead inherent in information collection outweighs the benefits, since the communication network is static, processors are aware of all other processors, and no work is created dynamically. Surveys of dynamic load balancing can be found in [20], [28].

**9.3.2.1 Random Polling (RP).** When a processor runs out of work it sends a request to a randomly selected processor. This continues until the processor finds work or there is no more work in the system and termination is established. Each processor is equally likely to be selected. This is a totally distributed algorithm, and has no bottlenecks due to centralized control. One drawback is that the communication

overhead may become quite large due to the unpredictable number of random requests generated. Also, in the worst case, there is no guarantee that any of the idle processors will ever be requested for work, and effectively no load balancing may be achieved. Detailed analysis and some implementation results on random polling are treated by Sanders in [26], [27].

**9.3.2.2 Global Round Robin with Message Combining (GRR-MC).** The idea behind a global round robin is to make sure that successive work requests go to processors in a round robin fashion. For example, in a parallel system of  $N$  processors, if the first work request goes to processor 0, the second one will go to processor 1, such that the  $i$ th request will be sent to processor  $i \bmod N$ . All processors will have been polled for work in  $N$  requests. This scheme requires global knowledge of which processor, say  $T$ , is to be polled next. A designated processor acts as the global round robin manager, and keeps track of  $T$ . When a node needs work it will refer to the manager for the current value of  $T$ . Before responding to other queries the manager will increment  $T$  to  $T + 1 \bmod N$ . The node looking for work can then send a request for tasks to the processor whose identity is equal to  $T$ . A major drawback of this scheme is that as  $N$  grows and the work requests increase, the manager will become congested with queries, posing a severe bottleneck.

Kumar, Grama, and Rao [24] suggested that message combining be introduced to reduce the contention for access to the manager. Processors are organized in a binary spanning tree with the manager as the root, where each processor is a leaf of this tree. When a processor needs the value of  $T$  it sends a request up the spanning tree towards the root. Each processor at an intermediate node of the tree holds requests received from its children for some predefined time  $d$  before it propagates them up in one combined request. If  $i$  is the cumulative number of requests for  $T$  received at the root from one of its children, then  $T$  is incremented by  $i$  before a request from another child is processed. The value of  $T$  before it was incremented is percolated back down the tree through the child. Information about combined requests is kept in tables at intermediate tree nodes until they are granted, so that the correct value of  $T$  percolates down the tree. An example of global round robin with message combining is illustrated in Figure 9.4.

### 9.3.3 Termination Detection

Termination is part of the global state of a distributed system, because it depends on the global availability of work, as opposed to the work load of a single processor. The definition of global termination for this system implies that all processors are idle, and that there are no work transfer messages in transit. In other words, since no additional work is created on the fly, global termination has occurred when all computation is complete. An idle node is one which has no work, and is searching for work using one of the dynamic load balancing algorithms described above. A busy node is one which has work to perform. A processor can change its state from busy to idle only when it finishes its tasks, and from idle to busy only when it receives a work transfer message. Clearly, termination is a stable state, because if all processes are idle, and there are

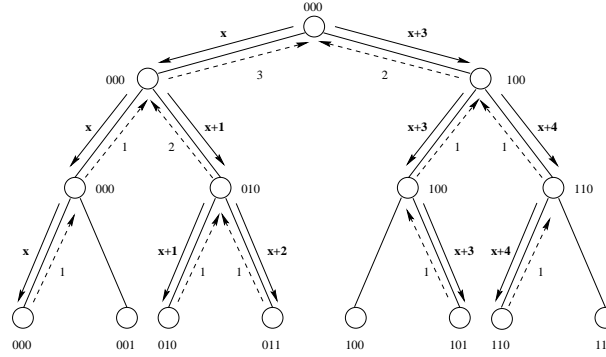


Figure 9.4 GRR-MC on a spanning tree, where  $x$  is the value of  $T$ , and  $N = 8$ .

no messages in transit, then no node will receive a work transfer message, and thus change its status to busy.

The global state of a distributed system can be captured in two ways, synchronously and asynchronously. The former can be achieved by freezing all processes on all processors and inspecting the state of each processor and each communication channel. This can be very time consuming when the number of processors is large, and more than one global state capture may need to be performed. Both of the algorithms described below establish termination asynchronously.

**9.3.3.1 Global Task Count (GTC).** The idea behind global task count is to keep track of all finished tasks, and use this to detect when all work has been performed, i.e., establish termination. This algorithm is applicable because the total number of tasks is available, and fixed. One processor, the manager, is responsible for keeping track of the finished tasks count  $C$ . Initially,  $C$  is set to 0. Whenever a processor completes its set of tasks, it sends a notification to the manager with the number of tasks that it completed. Upon receiving such a message the manager increments  $C$  accordingly. Eventually, as all has been performed, the value of  $C$  becomes equal to the known total number of tasks. At this point, the manager notifies all processors that termination has occurred.

Global task count detects termination immediately after it occurs, which makes it very fast. The total number of messages sent by a node to the manager is equal to the number of times that the node became busy with work, including its initial load. A potential drawback is that sending messages to a single manager may become a bottleneck, as the number of processors increases. On the other hand, total completion notifications are expected to be of order  $N$ , and to be spread out in time. This algorithm is very similar to Mattern's Credit Recovery Algorithm described in [30] and Huang's algorithm treated in [28]. Task count here corresponds to credits in Mattern's and Huang's algorithms that are not scaled to 1. The fact that credits are not scaled to 1 has the advantage of avoiding floating point representation issues that would otherwise be encountered. A separate proof of correctness for global task count is considered redundant.

**9.3.3.2 Token Passing (TP).** Token passing is a wave algorithm for a ring topology. For a thorough discussion of wave algorithms see [31]. A wave is a pass of the token around the ring where all processors have asynchronously testified to being idle. This is not enough to claim termination, since all nodes were polled at different times, and with dynamic load balancing, it is uncertain if they remained idle or later became busy. A second wave is needed to ascertain that there has been no change in the status of any processor. For a similar algorithm, see [6]. Termination is detected in at most two waves or  $2N$  messages, after it occurs. The total number of messages used depends on the total number of times the token is passed around the ring, but is bounded below by  $2N$ .

Each processor keeps track of its state in a local flag *IDLE*. Initially, *IDLE* is set to *false* since all processors start off with some load. Consequently, *IDLE* is set to *false* every time a processor receives more work as a result of dynamic load balancing. A token containing a counter  $T_c$  is being passed among all processors in a circular fashion. Upon receiving the token, a processor holds it until it has finished all its work, and/or before it is ready to send a work request. It is important to note that the sending of a request and the receiving of either a positive or negative reply are treated as an atomic unit. At that point, it checks the value of its *IDLE* flag. In case *IDLE* is *true* the processor increments the token counter  $T_c$  by 1. In the case where *IDLE* is *false*, the  $T_c$  is reset to 0, and the value of *IDLE* is set to *true*. After this, if the token counter happens to be equal to the number of processors  $N$ , termination is established and all processors are notified. Otherwise, the token is sent to the next processor in the ring.

## 9.4 PARALLEL IMPLEMENTATION

The distributed control algorithms were implemented in C to mesh with the existing analysis codes that were in both C and FORTRAN 77.

### 9.4.1 MPI

The Message Passing Interface (MPI) [29] is a message passing standard developed by the MPI forum—a group of more than 80 people representing universities, research centers, vendors and manufacturers of parallel systems. As a communications protocol MPI is platform independent, thread-safe, and has a lot of useful functionality—combining the best features of several existing messaging protocols [29]. A brief discussion of these attributes follows.

- **Platform independence:** MPI was developed to work on parallel platforms regardless of underlying architecture. This abstraction over native communication protocols makes MPI applications portable across architectures (distributed memory, shared memory, or network clusters) as long as an MPI implementation for the desired platform exists. For many architectures MPI implementations

are readily available, since the standard is widely supported by computer manufacturers.

- **Built-in functionality:** One of the important advantages of MPI is that it provides reliable communications, so the programmer does not have to deal with communication failures (see assumptions in Chapter 5.1). The standard also incorporates mechanisms for point-to-point and collective communication (e.g., broadcast, scatter, gather, etc.), overlapping computation and communication, process topologies and groups, and awareness and manipulation of the parallel environment. Much of this functionality has been incorporated in the parallel version of the code.
- **Thread-safety:** The dynamic load balancing code exploits a multi-threaded paradigm. This implies that all modules and packages used in the code have to be designed to work with threads, otherwise results are unpredictable.

### 9.4.2 Threads

Threads are distinct concurrent paths of execution within the same OS process that get scheduled within the allotted time of their parent process. Different scheduling techniques can be used depending on the package and the operating system support. For a discussion of threads see [28]. One of the challenging aspects of multithreaded design is that threads share access to their parent process' memory. This calls for mutual exclusion and synchronization techniques, like semaphores, monitors, etc. An advantage of this approach is that it exploits concurrency at the processor level. For example, a thread could be running on the I/O controller, another on the network service node, and a third one could be performing computations on the processing unit. Such concurrency can also be achieved by using nonblocking I/O or MPI calls, but organizing each logical task within the process in a thread can provide a finer-grained concurrency and a more intuitive design. For example, Kumar, Grama, and Rao show a state diagram describing a "generic parallel formulation" [24], where the grain of concurrency depends on the fixed unit of work. In a typical multithreaded approach threads that have no work stay idle without consuming processing cycles, and start working only when they are signaled that there is more work to be done, thus avoiding busy-wait. Overall, using threads with the MDO code introduced some complexities as a result of mutual exclusion and synchronization, but as a whole made it easier to design and implement the algorithms. The package used was POSIX threads, and mutual exclusion was implemented using semaphores.

## 9.5 DISCUSSION OF RESULTS

Figures 9.5–9.7 show snapshots of the states of nodes during execution for a small sample problem on  $N = 7$  nodes. Processors spend time performing file I/O, computation, and sitting idle. When dynamic load balancing is not in effect (Figure 9.5), processors spend half of their time being idle. With GRR-MC (Figure 9.6) and RP (Figure 9.7) idle states are more scattered, and are significantly reduced. It can also be seen that



Figure 9.5 Snapshot with *nupshot* utility of static load distribution,  $N = 7$ .

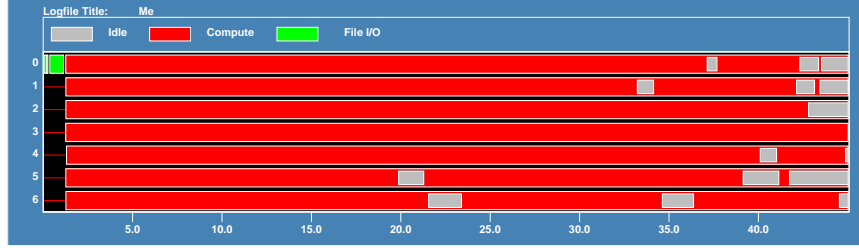


Figure 9.6 Snapshot with *nupshot* utility of GRR-MC with global task count termination,  $N = 7$ .

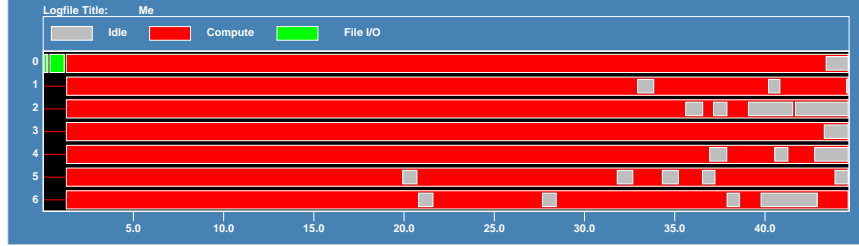


Figure 9.7 Snapshot with *nupshot* utility of RP with global task count termination,  $N = 7$ .

even though RP and GRR-MC result in different distributions, both are effective. To test scalability and efficiency a relatively large data set with approximately 2 million (2,026,231) designs was generated using the point selection algorithm described in Section 2.2 with order 4 and level 3. All runs were performed on Intel Paragon platforms, which have a mesh architecture with Intel i860 XP processors comprising the nodes.

Table 9.3 shows execution times from the Intel Paragon computer XP/S 7 (100 compute nodes) at Virginia Tech, and the Intel Paragon XP/S 5, XP/S 35, and XP/S 150 (128, 512, 1024 compute nodes, respectively) computers at the Oak Ridge National Laboratory Center for Computational Sciences. Times are given in hours, minutes, and seconds; for  $N \leq 64$  the average of five runs is reported, with the standard deviation in parentheses under the time (random polling uses the same fixed seed for all runs).

Table 9.3 Intel Paragon parallel times (hh:mm:ss) for low fidelity analysis of 2,026,231 HSCT designs.

$N$	RP		GRR-MC		Static
	GTC	TP	GTC	TP	
32	8:28:38 (6:38)	8:38:23 (2:33)	8:30:00 (1:30)	8:39:41 (:48)	12:48:11 (1:52)
64	4:08:05 (:03)	4:13:08 (:04)	4:13:08 (:02)	4:18:09 (:02)	7:06:59 (:09)
128	2:11:46	2:12:54	2:14:04	2:17:36	3:43:30
256	57:39	1:03:46	59:27	1:08:24	1:54:58
512	31:16	33:04	33:34	34:26	1:01:49
1024	15:20	15:43	17:30	17:27	29:26

For all other results,  $N \geq 128$ , only one run was completed because of limited access to larger Paragons. The table starts from  $N = 32$  nodes instead of  $N = 1$  because the time required to run 2 million designs on one processor is prohibitive ( $> 400$  hours). Furthermore, the current implementation generates all the PBIB designs in one chunk, so the memory required for storing all designs would also be prohibitive. Times in Table 9.3 do not include disk I/O (1.2GBytes) for the final results.

>From Figures 9.8 and 9.9 and Table 9.3 several observations can be made.

1. **Scalability:** all algorithms, including static distribution, scale well for  $N \leq 1024$  nodes, with random polling showing no noticeable degradation in efficiency at  $N = 1024$  nodes.
2. **Dynamic vs. static:** both dynamic load balancing techniques seem to be very effective, 35 to 50 percent better than static distribution, and this difference increases with the number of processors  $N$ .
3. **Stable execution times:** the standard deviations of total execution time for runs on 32 and 64 processors are very small, which indicates that performance is relatively stable. Random polling is expected to have more variance in execution time when the seed is not fixed, but not a significant difference.
4. **Superlinear speedup:** the latter rows of Table 9.3 exhibit superlinear speedup for global round robin with message combining and random polling. This implies that at 32 nodes the memory requirement (18 MBytes) for working with a relatively large number of tasks ( $\approx 63, 319$ ) per node can degrade performance on the XPS/7 platform due to resource starvation.
5. **Global task count vs. token passing:** Global task count seems to outperform token passing for  $N \leq 1024$  with GRR-MC and RP, but the relative difference decreases for larger  $N$ . Clearly, the overhead involved in processing all completion messages ( $\geq N$ ) by one manager node under global task count increases with  $N$ .

6. **Random polling vs. global round robin with message combining:** With both termination detection schemes random polling clearly outperforms global round robin with message combining. The relative difference in execution times increases as  $N$  becomes larger. The simplicity of the random polling algorithm leads to the lack of any significant overhead. Contention conditions are unlikely to occur because messages are randomly directed and typically the number of unsuccessful work acquisition messages increases significantly only just before termination. Global round robin with message combining, on the other hand, involves a longer wait, comprised of a fair number of communication messages across the spanning tree, before it can send a work acquisition request and the price of unsuccessful work acquisition requests is higher, because they imply more time spent idle. Furthermore, on average the total number of messages processed by a node running GRR-MC is higher than that for a node running RP, since each request is propagated back and forth through as many as  $\log_2 N$  other nodes. Finally, for both algorithms, the fact that all nodes start off with some load, which is expected to be relatively balanced among them for a large number of randomly long tasks, serves to decrease the initial number of unsuccessful work acquisition requests, which in turn improves performance.

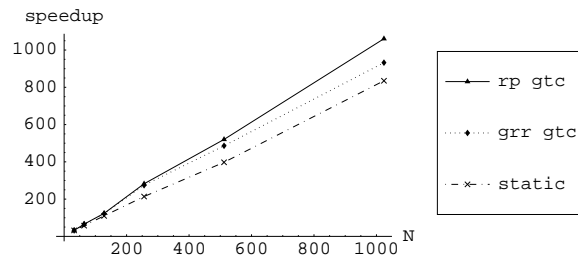


Figure 9.8 Speedup results for global task count using  $N = 32$  as the base.

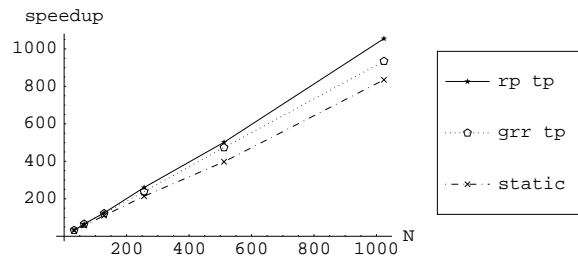


Figure 9.9 Speedup results for token passing using  $N = 32$  as the base.



## 9.6 SUMMARY

Distributed control and load balancing techniques were applied to an aspect of the multidisciplinary design optimization of a high speed civil transport. Two dynamic load balancing algorithms (random polling and global round robin with message combining) together with two necessary termination detection schemes (global task count and token passing) were implemented for the reasonable design space identification paradigm. Performance was evaluated on up to 1024 processors for all combinations of dynamic load balancing and termination detection schemes, plus the static distribution case. The effect of various algorithmic parameters was also explored, and found to be negligible except at extreme values. The results were very encouraging in terms of the effectiveness of dynamic load balancing (35–50 percent improvement over a static distribution), and the scalability of the algorithms (speedup was essentially linear). Most importantly, the time spent identifying the reasonable design space has been dramatically decreased, permitting the low fidelity analysis of 2 million designs, which was impractical before. The logical next step is to go beyond merely identifying the reasonable design space, and to identify good design regions within the reasonable design space, which would then be passed off to mildly parallel machines (e.g., IBM SP/2 or SGI Origin 2000) for “local” high fidelity optimization.

This effort is a stepping stone towards the goal of a MDO problem solving environment that will provide a complete and convenient computing environment for interactive multidisciplinary aircraft design. As shown by the experience of Burgee et al. [7, 14] and many others, some crucial disciplinary analysis codes (for structural mechanics, fluid dynamics, aerodynamic analysis, propulsion, to name a few) perform very poorly in a multidisciplinary parallel computing environment. These codes represent hundreds of man-years of experience and development, and are unlikely to be rewritten for parallel machines any time soon. Thus the challenge is to find approaches to MDO (e.g., variable complexity modeling and response surface techniques) which permit the use of massively parallel computing for some phases of the process (one such phase was demonstrated here) and legacy disciplinary codes on serial computers for other phases. One highly touted solution is “network computing”, but that still remains far from practical for serious large-scale multidisciplinary design.

## Acknowledgments

This work was supported in part by Air Force Office of Scientific Research grant F49620-92-J-0236, National Science Foundation grant DMS-9400217, and National Aeronautics and Space Administration grant NAG-2-1180. The authors gratefully acknowledge the use of the Intel Paragon XP/S 5, XP/S 35, and XP/S 150 computers, located in the Oak Ridge National Laboratory Center for Computational Sciences (CCS), funded by the Department of Energy’s Mathematical, Information, and Computational Sciences (MICS) Division of the Office of Computational and Technology Research.

## References

- [1] Balabanov, V., Kaufman, M., Giunta, A. A., Grossman, B., Mason, W. H., Watson, L. T., and Haftka, R. T. (1996). Developing customized weight function by structural optimization on parallel computers. In *37th AIAA/ASME/ASCE/AHS/ASC, Structures, Structural Dynamics and Materials Conference*, pages 113–125, Salt Lake City, UT, AIAA Paper 96–1336 (A96–26815).
- [2] Becker, J. and Bloebaum, C. (1996). Distributed computing for multidisciplinary design optimization using java. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 1583–1593, Bellevue, Washington. AIAA.
- [3] Bischof, C., Green, L., Haigler, K., and Jr., T. K. (1994). Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Fifth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization*, pages 73–86, Panama City, Florida. AIAA.
- [4] Box, G. E. P. and Behnken, D. W. (1960). *Some New Three Level Designs for the Study of Quantitative Variables*, volume 2. Technometrics.
- [5] Braun, R. and Kroo, I. (1995). *Development and application of the collaborative optimization architecture in a multidisciplinary design environment*, pages 98–116. In *Multidisciplinary Design Optimization: State of the Art*, N. Alexandrov, M.Y. Hussaini (Eds.). SIAM, Philadelphia.
- [6] Brzezinski, J., H  lary, J.-M., and Raynal, M. (1993). Distributed termination detection: General model and algorithms. Technical Report BROADCAST#TR93-05, ESPRIT Basic Research Project BROADCAST.
- [7] Burgee, S., Giunta, A. A., Balabanov, V., Grossman, B., Mason, W. H., Narducci, R., Haftka, R. T., and Watson, L. T. (1996). A coarse-grained parallel variable-complexity multidisciplinary optimization paradigm. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(4):269–299.
- [8] Dennis Jr., J., Lewis, R. M. (1994). Problem formulations and other issues in multidisciplinary optimization. Tech. Rep. CRPC–TR94469, CRPC, Rice University.
- [9] Dennis Jr., J., V. T. (1991). Direct search methods on parallel machines. *SIAM Journal of Optimization*, 1(4):448–474.
- [10] Doorly, D., Peir  , J., and Oesterle, J. (1996). Optimisation of aerodynamic and coupled aerodynamic-structural design using parallel genetic algorithms. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 401–409, Bellevue, Washington. AIAA.
- [11] Eldred, M., Hart, W., Bohnhoff, W., Romero, V., Hutchison, S., and Salinger, A. (1996). Utilizing object-oriented design to build advanced optimization strategies with generic implementation. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 1568–1582, Bellevue, Washington. AIAA.
- [12] Ghattas, O. and Orozco, C. (1995). *A parallel reduced Hessian SQP method for shape optimization*, pages 133–152. In *Multidisciplinary Design Optimization: State of the Art*, N. Alexandrov, M.Y. Hussaini (Eds.). SIAM, Philadelphia.

- [13] Giunta, A. A. (1997). *Aircraft multidisciplinary design optimization using design of experiments theory and response surface modeling methods*. PhD thesis, Virginia Polytechnic Institute and State University.
- [14] Guruswamy, G. (1998). Impact of parallel computing on high fidelity based multidisciplinary analysis. In *Seventh AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 67–80, St. Louis, Missouri. AIAA.
- [15] Hale, M. and Craig, J. (1995). Use of agents to implement an integrated computing environment. In *Computing in Aerospace 10*, pages 403–413, San Antonio, Texas. AIAA.
- [16] Harris Jr., R. V. (1964). An analysis and correlation of aircraft wave drag. Technical Report TM X-947, NASA.
- [17] Hinkelman, K. (1994). *Design and analysis of experiments*. John Wiley & Sons, Inc.
- [18] Hopkins, D., Patnaik, S., and Berke, L. (1996). General-purpose optimization engine for multi-disciplinary design applications. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 1558–1565, Bellevue, Washington. AIAA.
- [19] Hulme, K. F. and Bloebaum, C. (1996). Development of CASCADE: a multidisciplinary design test simulator. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 438–447, Bellevue, Washington. AIAA.
- [20] Kameda, H., Li, J., Kim, C., and Zhang, Y. (1997). *Optimal Load Balancing in Distributed Computer Systems*. Springer-Verlag.
- [21] Kaufman, M. D. (1996). Variable-complexity response surface approximations for wing structural weight in hsct design. Master's thesis, Virginia Polytechnic Institute and State University.
- [22] Knill, D., Giunta, A., Baker, C., Grossman, B., Mason, W., Haftka, R., and Watson, L. Response surface models combining linear and euler aerodynamics for hsct design. *Journal of Aircraft* (to appear).
- [23] Kroo, I., Altus, S., Braun, R., Gage, P., and Sobieski, I. (1994). Multidisciplinary optimization methods for aircraft preliminary design. In *Fifth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization*, pages 697–707, Panama City, Florida. AIAA.
- [24] Kumar, V., Grama, A. Y., and Vempaty, N. R. (1994). Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79.
- [25] Ridlon, S. (1996). A software framework for enabling multidisciplinary analysis and optimization. In *Fifth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization*, pages 1280–1285, Panama City, Florida. AIAA.
- [26] Sanders, P. (1994). A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 382–389, Kanazawa, Japan.

- [27] Sanders, P. (1995). Some implementations results on random polling dynamic load balancing. Technical Report iratr-1995-40, Universität Karlsruhe, Informatik für Ingenieure und Naturwissenschaftler.
- [28] Singhal, M. and Shivaratri, N. G. (1994). *Advanced Concepts in Operating Systems*. McGraw-Hill.
- [29] Snir, Marc, O., W., S., Huss-Lederman, S., Walker, D. W., and Dongarra, J. (1996). *MPI The Complete Reference*. The MIT Press.
- [30] Tel, G. (1991). *Topics in Distributed Algorithms*. Number 1 in Cambridge International Series in Parallel Computation. Cambridge University Press.
- [31] Tel, G. (1994). *Introduction to Distributed Algorithms*. Cambridge University Press.
- [32] Weston, R., Townsend, J., Edison, T., Gates, R. (1994). A distributed computing environment for multidisciplinary design. In *Fifth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization*, pages 1091–1095, Panama City, Florida. AIAA.
- [33] Wujek, B., Renaud, J., and Batill, S. M. (1995). *A concurrent engineering approach for multidisciplinary design in a distributed computing environment*, pages 189–208. In *Multidisciplinary Design Optimization: State of the Art*, N. Alexandrov, M.Y. Hussaini (Eds.). SIAM, Philadelphia.
- [34] Yoder, S. and Brockman, J. (1996). A software architecture for collaborative development and solution of mdo problems. In *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 1060–1062, Bellevue, Washington. AIAA.

# 10 NUMERICAL SEMICONDUCTOR DEVICE AND PROCESS SIMULATION ON SHARED MEMORY MULTIPROCESSORS: ALGORITHMS, ARCHITECTURES, RESULTS

Olaf Schenk<sup>†</sup>,

Klaus Gärtner<sup>‡</sup>, Bernhard Schmithüsen<sup>†</sup> and Wolfgang Fichtner<sup>†</sup> \*

<sup>†</sup> *Integrated Systems Laboratory,  
Swiss Federal Institute of Technology Zurich  
ETH Zurich, 8092 Zurich, Switzerland  
oschenk@iis.ee.ethz.ch*

<sup>‡</sup> *Weierstrass Institute for Applied Analysis and Stochastics  
Mohrenstr. 39, 10117 Berlin, Germany*

**Abstract:** We present PARDISO, a new parallel sparse direct linear solver for large-scale parallel semiconductor device and process simulations on shared memory multiprocessors. Since robust transient 2-D and 3-D simulations necessitate large computing resources, the choice of architectures, algorithms and their implementations becomes

---

\*The work of O. Schenk was supported by a grant from the Cray Research and Development Grant Program and the Swiss Commission of Technology and Innovation (KTI) under contract number 3975.1.

of utmost importance. Sparse direct methods are the most robust methods over a wide range of numerical properties and therefore PARDISO has been integrated into complex semiconductor device and process simulation packages. We have investigated popular shared memory multiprocessors and the most popular numerical algorithms commonly used for the solution of the classical drift-diffusion and the diffusion-reaction equations in process simulation. The study includes a preconditioned iterative linear solver package and our parallel direct linear solver. Moreover, we have investigated the efficiency and the limits of our parallel approach. Results of several simulations of up to 100'000 vertices for three-dimensional device simulations are presented to illustrate our approach towards robust, parallel semiconductor device and process simulation.

## 10.1 INTRODUCTION: SEMICONDUCTOR DEVICE SIMULATION AND ALGORITHMS

Numerical semiconductor device and process simulation is based on the solution of a coupled system of non-linear partial differential equations (PDE's) that can be either stationary, time dependent, or even complex with continuous dependencies of a parameter, depending on the problem considered [9, 10]. The nonlinear nature of the semiconductor transport equations, with exponential relations between variables, leads to discretized equations and the sparse linear systems are typically ill-conditioned. These systems are solved by *iterative methods* or *direct methods*.

Generally, the memory requirements of an iterative method are fixed, known in advance, and these methods require less main memory than direct methods. Heiser *et al.* [8] compared sparse direct and iterative algorithms for semiconductor device simulations and expected main memory requirements for sparse direct methods in the 10-100 Gbytes range for a simulation with 100'000 vertices and 300'000 unknowns. However, they used only local fill-in reduction methods like minimum-degree based algorithms [17] and the sparse direct solver did not exploit the memory hierarchy of the computing architecture.

The rapid and widespread acceptance of shared memory multiprocessors, from the desktop to the high-end server, has now created a demand for parallel device and process simulation on such shared memory multiprocessors. In this paper, we present a study of large-scale semiconductor device and process simulations and compare the time requirements of complete simulations using iterative methods and our parallel direct solver. PARDISO exploits the memory hierarchy of the architecture using the clique structure of the elimination graph by supernode algorithms, thus improving memory and processor locality. The reordering features state-of-the-art techniques, e.g. multilevel recursive bisection [13, 14], for the fill-in reduction. The combination of block techniques, parallel processing, and global fill-in reduction methods for three dimensional semiconductor devices results in a significant improvement in computational performance.

In section 10.2 we describe in detail the algorithms used in the parallel sparse LU solver. Furthermore, we provide an overview of the shared memory multiprocessors (SMPs) and present the numerical results on these SMPs. Section 10.3 provides the comparison of iterative methods and sparse direct methods for several semiconductor

device simulation examples on different architectures. The scalability is demonstrated for the simulator `DESSIS-ISE` with PARDISO for a realistic 3-D example with 105'237 vertices and 315'711 unknowns. Section 10.4 contains a summary of our results.

## 10.2 HIGH COMPUTATIONAL PERFORMANCE FOR PARALLEL SPARSE LU FACTORIZATION

In 3-D semiconductor process and device simulation, typical values for the grid size range from several thousand up to several hundred thousand vertices, today. If three non-linear partial differential equations are solved in a fully coupled scheme, the number of unknowns is up to 300'000. The linear solver usually dominates the CPU times of the simulation on a single-processor architecture. It is, therefore, mandatory to use the fastest methods and implementation available.

Regarding the numerical stability of sparse Gaussian elimination it is well known that the method is guaranteed to be numerically stable when symmetric positive definite systems are solved [7]. In the area of semiconductor device simulation the sparse matrices resulting from the discretization of the drift-diffusion equations are unsymmetric. If such an unsymmetric matrix results in an unstable elimination process, one usually applies pivoting strategies to cure the problem. However, it is our experience that one can often solve these unsymmetric systems with acceptable accuracy with restricted pivoting. Hence, the non-symmetric structure of  $A$  is extended to a structural symmetric one and the reordering can be computed in advance. The experience clearly shows that the structured symmetric approach with pivoting in dense diagonal blocks is the most favourable one for our class of problems — if it fails due to the pivoting restrictions, the simulator perturbs the matrix.

We find that it is important to tackle three different issues for an efficient sparse LU direct solver. The first aim is to reduce the fill-in during the elimination process, the second one is a high single-node performance, and the third goal high parallel efficiency on a shared memory multiprocessor.

### 10.2.1 Reordering: minimum degree and nested dissection

Due to the restricted pivoting scheme we can now concentrate on the reordering for structurally symmetric linear systems. From the theory of sparse direct matrices [4, 6] it is well known that reordering has a drastic effect on the time and the memory requirements of the LU factorization of the sparse matrix  $A$ . The reordering is typically solved by two competing types of methods — *minimum degree algorithms* and *nested dissection algorithms*.

The minimum degree algorithm is a local algorithm. At each step it eliminates that nodes with the smallest degree since this is a heuristics that often minimizes fill-in locally for planar graphs. The nested dissection algorithm is a global algorithm. At each step a vertex separator of the graph is found which splits the graph into two or more disconnected components. The vertex separator is ordered last, since this

guarantees a minimal fill-in at the end of the factorization. In recent years, significant progress has been made in graph partitioning and reordering algorithms — especially with respect to the nested dissection algorithm for sparse matrices. Among the many heuristics proposed for approximately solving the NP-hard combinatorial problem is the recursive spectral bisection method first proposed by Pothén, Simon and Liou [20].

We have investigated several heuristics for the minimum degree and the nested dissection algorithm. Up to now, the only ones we find effective for our problem are the multiple minimum degree algorithm [17] and the recursive bisection algorithm implemented into METIS [13], a popular package that delivers high quality reorderings for sparse matrices. Table 10.1 shows the quality of the reordering with both types of fill-in reduction methods for typical semiconductor device and process simulation matrices.

*Table 10.1* Comparison of multiple minimum degree (MMD) and recursive nested dissection (METIS) for 2-D and 3-D semiconductor device and process simulation matrices

Matrix	Rows	Nonzeros	$ LU / A $	
			MMD	METIS
1 2D eth-points	151'389	1'046'105	5.62	6.04
2 2D eth-load.bic	56'941	393'415	6.44	6.69
3 2D ise-mosfet-1	12'024	250'740	6.11	6.24
4 2D ise-mosfet-2	24'123	504'765	6.47	6.66
5 3D eth-eeeprom	12'002	630'002	13.12	10.48
6 3D ise-igbt-coup	18'668	412'674	24.07	18.45
7 3D eth-eclt	25'170	1'236'312	20.44	14.49
8 3D ise-soir-coup	29'907	2'004'323	20.69	14.14
9 3D eth-mosfet	31'789	1'633'499	23.48	16.36
103D eth-eclt-big	59'648	3'225'942	33.06	19.77

Comparing the fill-in for the sparse LU factorization with the two popular fill-in reduction algorithms allows us to draw the conclusion that the multiple minimum degree algorithm is only suitable for two-dimensional grids. The nested dissection approach of the METIS package results in much smaller fill-in for three-dimensional matrices from semiconductor device and process simulations.

### 10.2.2 High single node performance with block algorithm

The key idea behind the factorization for sparse linear systems is based on the concept of a supernode [1, 18]. During the symbolic factorization, supernodes are identified as a set of contiguous columns in the factor L that share the same sparsity structure below the dense triangular block. The supernode block numerical factorization operates



on groups of columns at the same time and involves mainly dense matrix-matrix multiplications. The main effect is a strong reduction of memory traffic — resulting in a significant improvement of computational performance [19, 23, 3].

### 10.2.3 Parallel sparse LU factorization on shared memory multiprocessors

Ng and Peyton [19] have introduced a parallel sparse supernode Cholesky algorithm on a shared-memory vector supercomputer Cray Y-MP. They implemented a parallel left-looking supernode algorithm that schedules the supernode tasks dynamically on the available processors. The number of interprocess synchronizations required in that Cholesky factorization algorithm is proportional to the number of *compressed subscripts* [24]. The number of compressed subscripts is precisely the number of off-diagonal nonzero entries in the last column of all supernodes. Furthermore, the left-looking algorithm proposed in [19] is implemented with `saxpy` operations in the terminology of the BLAS [15] to allow supervector speed on the Cray supercomputer.

Based on the observation that `saxpy` operations are not suitable for BLAS level-3 speed on RISC multiprocessor architectures and that the interprocess communication is an important factor, we devised an algorithm that is close to optimal in computational performance for the class of matrices studied. The main features of our approach are that it is based on a panel partitioning technique, it reduces the communication overhead to order  $O(n)$ , and it is completely independent of the reordering. The algorithm with reduced synchronization is shown in Figure 10.1.

To obtain a moderate load balancing on SMPs we introduce panels at the end of the elimination process. Panels are fractions of supernodes and the factorization of panels makes the computational kernels highly efficient. The synchronization is organized within two critical sections, code segments that can be executed by only one thread at a time. A scheduler dynamically assigns a sub-supernode to a thread. There is no notion of ownership of sub-matrices by processors - the assignment is completely dynamic, only depending on the execution speed of each individual processor. After the assignment, the thread executes the first critical section and removes all previously computed external sub-supernodes from the queue. The external supernode updates are now completely independent from other tasks resulting in a high level of concurrency. Having performed all external updates, the thread computes the internal supernode LU factorization. As soon as the factorization of this supernode is finished, the supernode is marked as ready. The thread enters the second critical section and marks all parent sub-supernodes. In terminology of *left and right looking* [4, 19] both techniques are used for this interplay and the algorithm introduces order  $O(n)$  critical locks.

### 10.2.4 Shared memory multiprocessor systems

The technical data of the multiprocessor architectures used in the benchmarks are gathered in Table 10.2 and Table 10.3. In this study we consider two different shared memory architectures: shared memory parallel vector supercomputers, a 12-CPU NEC

```

1:  $Q \leftarrow \emptyset$ 
2: Scheduler = {bottom-up traversal of the elimination tree}
3: for P=1, #proc
4:   while (Scheduler not empty)
5:     lock I
6:     Scheduler  $\leftarrow$  Scheduler  $\setminus \{J\}$ 
7:     /* panel J is computed by process P */
8:     unlock I
9:     while  $\exists$  panel K in  $Q$  with  $A_{K,J} \neq 0$ 
10:      /* outer factorization, dependent on other
11:        processes */
12:      lock II
13:      remove all computed descendants K of panel J
14:      from queue  $Q$  /* left looking */
15:      unlock II
16:       $A_{*,J} = A_{*,J} - A_{*,K} * A_{K,J}$ 
17:       $A_{J,*} = A_{J,*} - A_{J,K} * A_{K,*}$ 
18:    end while
19:    /* inner factorization, independent of other
20:      processes */
21:     $A_{J,J} =: L_{J,J} * U_{J,J}$ 
22:     $A_{*,J} = A_{*,J} * U_{J,J}^{-1}$ 
23:     $A_{J,*} = L_{J,J}^{-1} * A_{J,*}$ 
24:    lock II
25:    mark all parents K of panel J in queue  $Q$ 
26:    /* right looking */
27:    unlock II
28:  end while
29: end for

```

Figure 10.1 Parallel left-right looking sparse LU factorization

SX-4 and a 16-CPU Cray J90, and shared memory multiprocessor servers, an 8-CPU DEC AlphaServer 8400, an 8-CPU SGI Origin 2000, and an 8-CPU Sun Enterprise 4000. Table 10.3 summarizes the characteristics of the individual processors, including the clock speed, the peak Mflop rate, and the peak LINPACK performance. We use 64-bit-arithmetic for all factorizations on all multiprocessor architectures discussed in this paper.

### 10.2.5 Computational Performance

The experiments are carried out with the sparse matrices presented in Table 10.1. All linear systems have an irregular structure and the examples have up to 2% non-zeros entries in the factors  $L$  and  $U$ . Implementation issues are very important — the difference between a good and a poor implementation may reach factor of twenty or more. As we can see from the data presented in Figures 10.3 to 10.8, PARDISO delivers substantial

Table 10.2 Characteristics of the shared memory multiprocessor systems used in the benchmarks.

SMP Machine	Processor	CPUs	Bandwidth
Sun Enterprise 4000	UltraSPARC-II	8	1.3 GB/s
DEC AlphaServer 8400	EV5.6 21164	8	1.2 GB/s
SGI Origin 2000	MIPS R10000	64	780 MB/s
Cray PVP	J90	16	51.2 GB/s
NEC PVP	SX-4	12	256 GB/s

Table 10.3 Clock speed, peak floating point and LINPACK (1000x1000) performance of one processor.

Processor	Clock speed	Peak Mflop/s	LINPACK Mflop/s
UltraSPARC-II	336 MHz	672	461
Alpha EV5.6 21164	612 MHz	1224	764
MIPS R10000	195 MHz	390	344
J90	100 MHz	200	202
SX-4	125 MHz	2000	1929

computational performance on supercomputers and high-end workstations and servers. For larger sparse matrices, we see that the package achieves approximately 220 Mflop/s on a one-processor SGI ORIGIN 2000. Compared to the vendor-optimized LINPACK, this translates to roughly 68% of that performance. This is the consequence of using level-3 BLAS efficiently. Table 10.2 shows the influence of the new synchronization scheme. The left-right looking supernode algorithm decreases the amount of synchronization events to only  $O(n)$  and it delivers substantial speedup, even for moderate problem sizes.

### 10.3 COMPARING ALGORITHMS AND MACHINES FOR COMPLETE SEMICONDUCTOR DEVICE SIMULATIONS

For our comparison in Table 10.4, we used five typical semiconductor device simulation cases: a 2-D transient simulation with a hydrodynamic model and 14'662 vertices, and four 3-D quasistationary device simulations, one with 10'870 vertices, one with

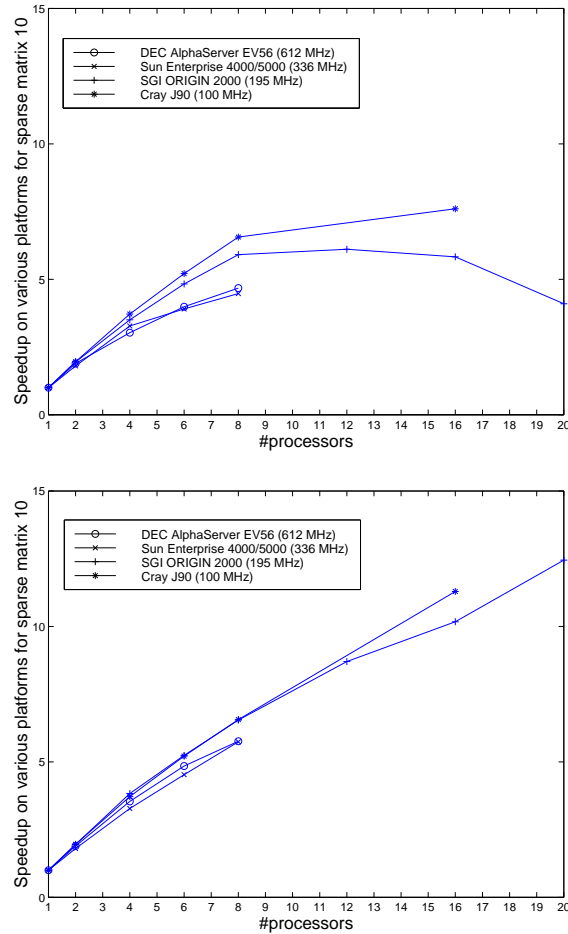


Figure 10.2 Speedup on various platforms for matrix eth-eclt-big with the left-looking approach (upper, synchronization events proportional to number of compressed subscripts) and the left-right looking approach (lower, synchronization events proportional to  $O(n)$ ).

12'699 vertices, one with 26'855 vertices and one with 105'237 vertices. All grids are highly irregular and the linear systems are very sparse. The irregularity of the grids leads to more complicated structures of the linear systems that must be solved during the simulation. The machines are shared memory multiprocessors and the processor characteristics are summarized in Table 10.3. We used an 32-processor SGI Origin 2000 with sixteen Gbytes of main memory, an eight-processor DEC AlphaServer with two Gbytes, and an eight-processor SUN Enterprise 4000 with two Gbytes of main memory.

Comparing the performance of two sparse direct solvers on the same machine with one single processor, we see that in all cases PARDISO [22] is significant faster than

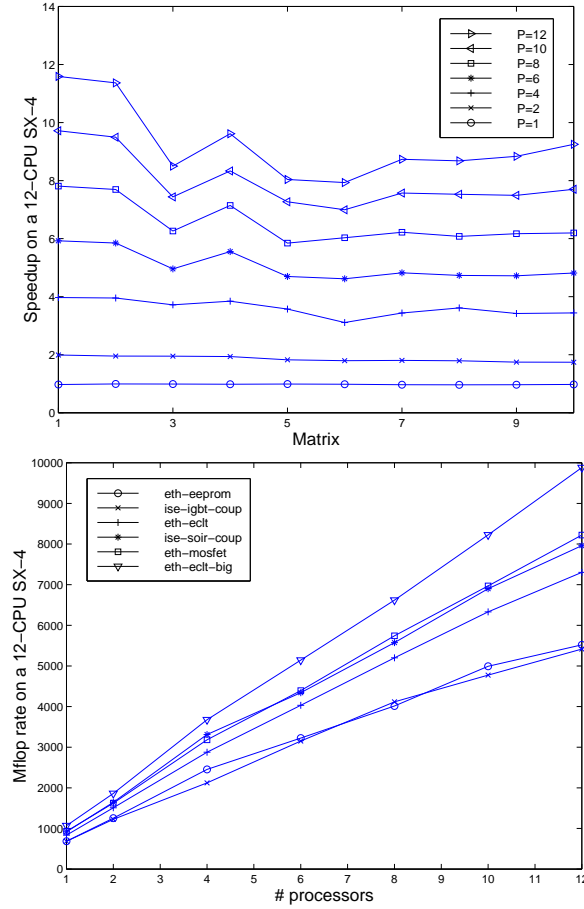


Figure 10.3 Speedup and Mflop/s on a 12-CPU NEC SX-4

$SUPER_{ISE}$  [16, 12].  $PARDISO$  uses the memory hierarchy between cache and main memory significantly better, and the improvement of the reordering is especially large for 3-D grids. Table 10.4 clearly reflects the performance difference due to algorithmic improvements.

On the other side, preconditioned sparse iterative methods play an important role in the area of semiconductor device and process simulations. Of the wide variety of iterative solvers available in the literature [2, 21] it has been found that only a few are capable of solving the equations of large semiconductor device and process simulations. This is mainly due to the bad conditioning of the matrices in the coupled solution scheme. So far, BiCGstab(2) [25] with a fast and powerful incomplete LU decomposition preconditioning and reverse Cuthill-McKee ordering appears to be the best choice for solving large sparse linear systems [5, 11]. The default preconditioning in the package  $SLIP90_{ISE}$  is  $ILU(5, 1E-2)$ . For increasingly difficult linear systems, the

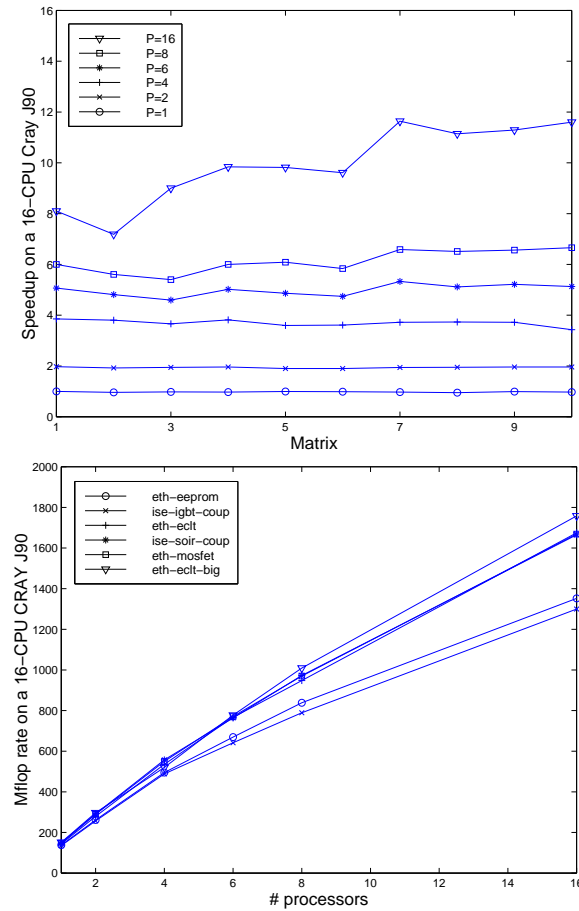


Figure 10.4 Speedup and Mflop/s on a 16-CPU Cray J90

fill-in level and the threshold value for dropping a fill-in entry in the ILU decomposition is changed in five steps to ILU(20,1E-5).

However, the results for the linear solver have to be seen in the context of the complete simulation task. One often observed problem of the iterative methods used is the strong reduction of the average step size during a quasistationary or transient simulation due to convergence problems. The typical time step size for non trivial examples is in the range of 0.01 to 0.001 times that found for a direct method. Furthermore, looking at the quasistationary 3-D simulation with  $105 \times 237$  vertices in Table 10.4 we see that the iterative solver failed during the simulation. The main memory requirement of DESSIS<sub>ISE</sub> with the parallel direct solver was about 4 Gbytes on the SGI Origin 2000 for the largest example with  $105 \times 237$  vertices.

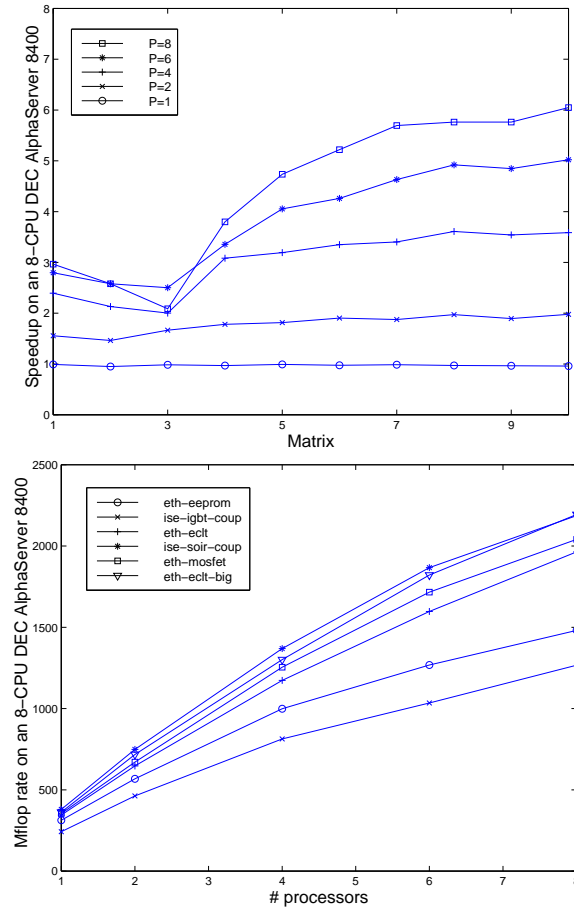


Figure 10.5 Speedup and Mflop/s on an 8-CPU DEC AlphaServer 8400

Up to now, function evaluation and Jacobian computation are not parallelized; likewise, the reuse of factorized matrices as preconditioners in a CGS iteration (a feature of PARDISO and controlled by measured times automatically) is not activated by the simulation programs. Hence the cross over point from direct to iterative methods for the problem class is on bandwidth limited RISC machines well above  $10^4$  vertices. For moderately sized problems the parallelization of the assembling of the linear equations would reduce the time consumption (compare Table 10.5). For the device simulator  $DESSIS_{ISE}$ , which is written in the object oriented language C++, it is natural to start the task after the announcement of OpenMP for C++.

The Table 10.6 shows two semiconductor device simulation cases on a 32 processor SGI Origin 2000. The solution of the sparse linear systems dominates the execution time of the serial implementation especially for the 3D grid with  $105'237$  unknowns.

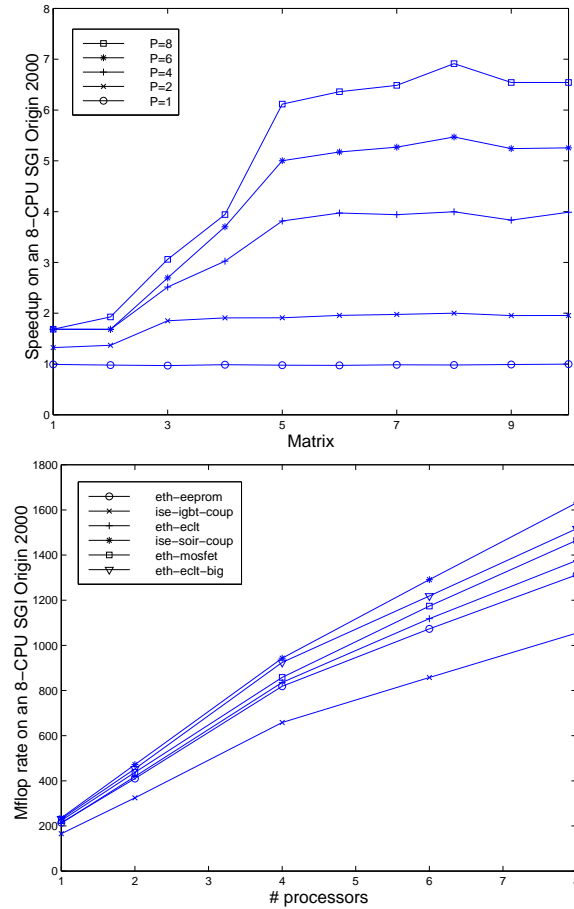


Figure 10.6 Speedup and Mflop/s on an 8-CPU SGI Origin 2000

The numerical results demonstrate scalability for for todays very large semiconductor device simulations.

## 10.4 CONCLUSION

We have presented PARDISO, a parallel direct solver for robust parallel semiconductor device and process simulation. The implementation features state-of-the-art techniques. In order to perform large scale parallel simulations, we have investigated popular shared memory multiprocessors. A high level of sequential and parallel efficiency has been achieved on typical shared memory parallel servers and supercomputers. The cross over point from direct to iterative methods for semiconductor process and device simulation on bandwidth limited RISC machines well above  $10^4$  vertices. For larger problem sizes up to 100'000 vertices and 300'000 unknowns 32



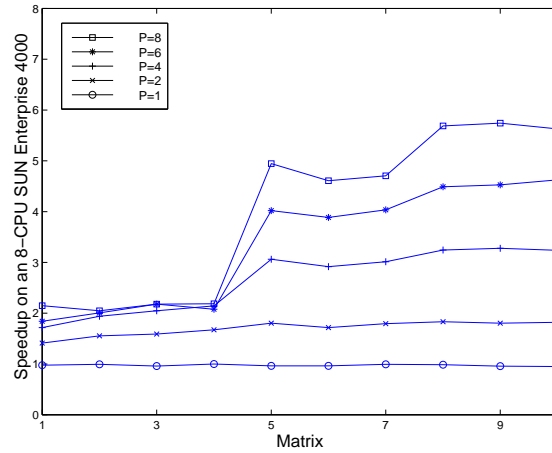


Figure 10.7 Speedup on an 8-CPU Sun Enterprise 4000

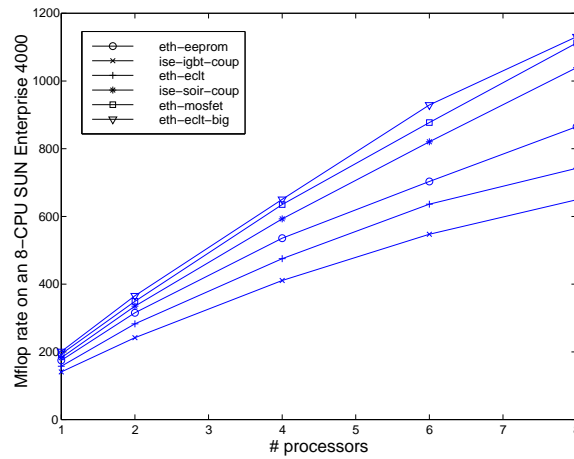


Figure 10.8 Mflop/s on an 8-CPU Sun Enterprise 4000

processor machines can be used efficiently due to the parallel direct linear solution algorithm. Tangible benefits can be achieved on today's eight-processor workgroup servers. However, the gap between some  $10^4$  and  $10^6$  vertices may be bridged by multigrid methods.

## Acknowledgments

The authors appreciate the fruitful discussions with Esmond Ng (Oak Ridge National Laboratory). We thank Igor Zacharov (Silicon Graphics) for his efforts to perform

Table 10.4 Comparison of different algorithms and packages on high-end servers. Wall clock times in hours for one complete semiconductor device simulation with DESSIS<sub>ISE</sub> on one processor.

	SLIP90 iterative	SUPER direct	PARDISO direct
<i>2-D 14'662 vertices, six nonlinear PDE's</i>			
DEC Alpha	17.0	25.2	11.1
SUN Enterprise	37.0	55.0	24.4
SGI Origin	29.3	44.2	19.5
<i>3-D 10'870 vertices three nonlinear PDE's</i>			
DEC Alpha	2.3	2.5	0.7
SUN Enterprise	2.2	3.9	1.3
SGI Origin	4.1	7.0	1.9
<i>3-D 12'699 vertices three nonlinear PDE's</i>			
DEC Alpha	1.1	5.4	0.9
SUN Enterprise	1.9	7.6	1.5
SGI Origin	1.3	6.0	1.2
<i>3-D 26'859 vertices three nonlinear PDE's</i>			
DEC Alpha	3.2	no mem.	5.3
SUN Enterprise	7.0	no mem.	11.7
SGI Origin	5.4	209.4	9.0
<i>3-D 105'237 vertices three nonlinear PDE's</i>			
DEC Alpha	failed	no mem.	no mem.
SUN Enterprise	failed	no mem.	no mem.
SGI Origin	failed	no mem.	401.1

the 32-processor parallel DESSIS-<sub>ISE</sub> benchmark on the SGI Origin 2000. Finally we want to mention the superb support within the CRAY-ETH Zurich SuperCluster cooperation.

## References

- [1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *The International*

Table 10.5 Decomposition of the Execution Time of the Serial DESSIS<sub>ISE</sub> simulation with PARDISO into assembling time (Rhs-time, Jacobian-time) and solution time for the sparse linear systems (Solve-time) on one processor.

	Rhs-time	Jacobian-time	Solve-time
2-D 14'662 vertices, six nonlinear PDE's	13 %	21 %	64 %
3-D 10'870 vertices three nonlinear PDE's	27 %	7 %	64 %
3-D 12'699 vertices three nonlinear PDE's	15 %	3 %	82 %
3-D 26'859 vertices three nonlinear PDE's	32 %	2 %	64 %
3-D 105'237 vertices three nonlinear PDE's	1 %	1 %	98 %

Table 10.6 Parallel DESSIS<sub>ISE</sub> semiconductor device simulator on shared memory multiprocessor SGI Origin 2000. Wall clock times in hours for one complete simulation.

# processors	3-D 12'699 vertices	3-D 105'237 vertices
SGI Origin 2000		
1	1.20	401.00
2	0.75	202.11
4	0.51	103.06
6	0.46	69.34
8	0.43	53.13
30	-	16.31

*Journal of Supercomputer Applications*, 1(4):10–30, 1987.

- [2] R. Barrett, 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, 1994.
- [3] J. Demmel, J. Gilbert, and X. Li. An asynchronous parallel supernodal algorithm to sparse partial pivoting. Technical Report CSD-97-943, University of California at Berkeley, Feb. 1997, 1997. To appear in SIAM J. Matrix Anal. Appl.

- [4] I. Duff, A. Erisman, and J.K.Reid. *Direct methods for sparse matrices*. Oxford Science Publications, 1986.
- [5] D. Fokkema. *Subspace methods for linear, nonlinear, and eigen problems*. PhD thesis, Utrecht University, 1996.
- [6] A. George and J. Liu. *Computer solution of large sparse positive definite systems*. Prentice-Hall, 1981.
- [7] G. Golub and C. van Loan. *Matrix computations*. The John Hopkins University Press, 3rd edition, 1996.
- [8] G. Heiser, C. Pommerell, J. Weis, and W. Fichtner. Three-dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Transactions on Computer-Aided Design*, 10(10):1218–1230, 1991.
- [9] Integrated Systems Engineering AG. *DESSIS-ISE Reference Manual*. ISE Integrated Systems Engineering AG, 1998.
- [10] Integrated Systems Engineering AG. *DIOS-ISE Reference Manual*. ISE Integrated Systems Engineering AG, 1998.
- [11] Integrated Systems Engineering AG. *SLIP90-ISE Reference Manual*. ISE Integrated Systems Engineering AG, 1998.
- [12] Integrated Systems Engineering AG. *SUPER-ISE Reference Manual*. ISE Integrated Systems Engineering AG, 1998.
- [13] G. Karypis and V. Kumar. Analysis of multilevel graph algorithms. Technical Report MN 95 - 037, University of Minnesota, Department of Computer Science, Minneapolis, MN 55455, 1995.
- [14] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report MN 98 - 019, University of Minnesota, Department of Computer Science, Minneapolis, MN 55455, 1998.
- [15] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–325, 1979.
- [16] A. Liegmann. *Efficient Solution of Large Sparse Linear Systems*. PhD thesis, ETH Zürich, 1995.
- [17] J. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.
- [18] J. Liu, E. Ng, and B. Peyton. On finding supernodes for sparse matrix computations. Technical Report TM-11563, Oak Ridge National Laboratory, 1990.
- [19] E. Ng and B. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on Scientific Computing*, 14(4):761–769, 1993.
- [20] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis & Applications*, 11(3):430–452, July 1990.
- [21] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

- [22] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. Technical Report 98/40, Integrated Systems Laboratory, ETH Zurich, Swiss Fed. Inst. of Technology (ETH), Zurich, Switzerland, Submitted to BIT Numerical Mathematics, 1998.
- [23] O. Schenk, K. Gärtner, and W. Fichtner. Parallel sparse LU factorization in a shared memory computing environment. In *The proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 907–914, 1998.
- [24] A. Sherman. *On the efficient solution of sparse linear and nonlinear equations*. PhD thesis, Yale University, New Haven, CT, 1975.
- [25] G. Sleijpen, H. van der Vorst, and D. Fokkema. BiCGSTAB(l) and other hybrid Bi-CG methods. Technical Report TR Nr. 831, Department of Mathematics, University Utrecht, 1993.



# 11 OVERLAPPING MESH TECHNIQUE FOR COMPRESSIBLE FLOWS - PARALLEL IMPLEMENTATION

Jacek Rokicki \*

*Zentrum für Hochleistungsrechnen, TU Dresden  
D-01062 Dresden, Germany*

rokicki@zhr.tu-dresden.de

Dimitris Drikakis †

*Department of Mechanical Engineering, UMIST  
PO Box 88, Manchester M60 1QD, United Kingdom*

drikakis@umist.ac.uk

Jerzy Majewski

*Institute of Aeronautics and Applied Mechanics, Warsaw University of Technology*

---

\*Permanent Address: Institute of Aeronautics and Applied Mechanics, Warsaw University of Technology, Nowowiejska 24, 00-665 Warsaw, Poland

†On move to Queen Mary and Westfield College, University of London, Engineering Department, London E1 4NS, UK

*Nowowiejska 24, 00-665 Warsaw, Poland*

jmajewsk@meil.pw.edu.pl

Jerzy Zóltak

*Aviation Institute**Al. Krakowska 110/114, Warsaw, Poland*

jzoltak@meil.pw.edu.pl

**Abstract:**

In this paper, parallelization of the Chimera overlapping-mesh technique and its implementation in conjunction with an implicit Riemann solver is presented. The parallelization of the method is based on the PVM approach. Computations are performed for compressible flows over multi-element airfoils. Efficiency results are presented for fairly complex domains consisting of a large number of meshes overlapping each other in an almost arbitrary manner, including multiple overlaps. The parallel performance of the method is investigated on the Cray CS6400 and Cray T3E computing platforms.

**11.1 INTRODUCTION**

Simulation of fluid flow in complex geometries continues to be one of the major research areas in Computational Fluid Dynamics (CFD). Several CFD methods have been developed over the last two decades, but improvement of their accuracy and efficiency remains a challenging problem. Nowadays, several different approaches can be employed to reduce the computational cost, including parallelization, implicit solvers and techniques for the acceleration of the numerical convergence e.g. multigrid.

The Overlapping Mesh Technique known also as the *Chimera method* or *Unstructured Domain Decomposition* (e.g. [1, 5, 8, 9]) can be employed as an efficient tool for grid generation in complex geometries, while more recently it has received particular attention due to its promising capabilities to handle moving boundaries [11].

This technique is based on the subdivision of the physical domain into overlapping subdomains (see a partition of flow domain around the GA(W)-1 airfoil in Fig. 11.1) and it is therefore inherently suitable for use on parallel computers. Subsequently, the system of flow equations is solved on each subdomain separately, and the global solution is obtained by iteratively adjusting the boundary conditions on each subdomain.



In the present study parallelization of the method has been obtained via the PVM approach. Investigation of the efficiency, including issues related to load balancing, has been performed for various grid topologies. Convergence studies are performed for subsonic and transonic flows around two-element airfoils.

## 11.2 IMPLICIT UNFACTORED SOLUTION

The compressible flow of an inviscid fluid is governed by the Euler equations. These equations can be written in dimensionless conservation form and for a general curvilinear coordinate system, as:

$$(JU)_t + E_\xi + G_\eta = 0. \quad (11.1)$$

where  $J$  is the Jacobian of the transformation from curvilinear  $(\xi, \eta)$  to Cartesian coordinates  $(x, y)$ ; the vector  $U$  contains the conservative variables  $U = (\rho, \rho u, \rho v, e)^T$  where  $\rho$  is the density,  $u, v$  are the Cartesian velocity components and  $e$  is the total energy per unit volume. The inviscid fluxes are denoted as  $E$  and  $G$ . The system of the governing equations is completed by the perfect gas equation of state.

The implicit discretization of equation (11.1) yields

$$J \frac{U^{n+1} - U^n}{\Delta t} + E_\xi^{n+1} + G_\eta^{n+1} = 0. \quad (11.2)$$

The system of equations (11.2) is solved by a Newton-type method. A third-order characteristics-based scheme [2, 3] is employed for discretizing the inviscid fluxes using the characteristic values of the conservative variables at the cell faces. A Godunov-type upwind scheme up to third-order accuracy has also been employed for the calculation of the characteristic cell face values.

## 11.3 THE OVERLAPPING-MESH ALGORITHM

The overlapping-mesh technique consists of the following steps

- decomposition of the physical domain into overlapping subdomains (see Fig. 11.1);
- advancement of the solution on each subdomain using the implicit Riemann solver;
- update of boundary conditions on the boundary interfaces using information from all local solutions.

The last two steps are performed iteratively until convergence is reached.

In the case of arbitrary overlap of subdomains, there exist two difficulties that need to be taken into account:

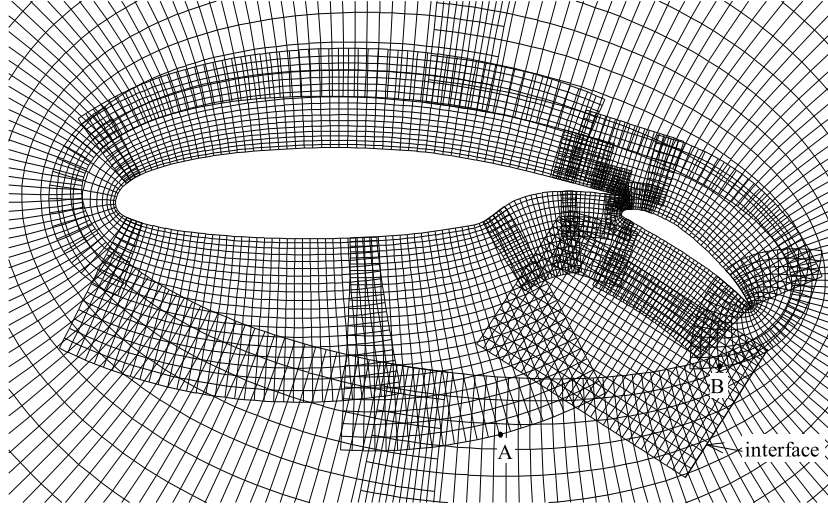


Figure 11.1 The system of 22 overlapping grids around the two-element GA(W)-1 airfoil

- in general the grid points do not coincide in the overlap region (see point A in Fig. 11.1);
- some interface points are overlapped by more than one subdomain (multiple-overlap) and it is not clear from which subdomain's solution the new boundary conditions should be generated (see point B in Fig. 11.1).

To overcome the first difficulty one needs to employ suitable interpolation procedure. The second difficulty can be tackled by combining all local solutions into a global solution using blending functions similar to those proposed in the references [8, 9] for incompressible flows. This combination of local solutions is used in turn to update the boundary conditions at the interfaces.

### 11.3.1 General structure of overlapping and grid generation

We consider here the general case in which the flow domain  $\Omega$  is fully covered by several overlapping subdomains  $\Omega_1, \dots, \Omega_K$ ,  $\Omega = \Omega_1 \cup \dots \cup \Omega_K$ . If  $\Gamma$  and  $\Gamma_j$  denote the boundaries of  $\Omega$  and  $\Omega_j$  subdomains, respectively, then  $\gamma_j = \Gamma_j \setminus \Gamma$  will denote the interface between the subdomains in which boundary information takes place.

In order to obtain convergence of the global solution, sufficient overlapping between subdomains must exist. This requirement can be expressed using the *internal distance* function  $d_j(A)$  [8, 9] defined as:

$$d_j(A) := \begin{cases} 0 & A \notin \Omega_j \\ \rho(A, \gamma_j) & A \in \Omega_j \end{cases} \quad (11.3)$$

where  $\rho(A, \gamma_j)$  stands for the distance between point  $A$  and the interface  $\gamma_j$ . The characteristic size of the overlap structure is defined as

$$\delta := \inf_{A \in \Omega} d(A) := \inf_{A \in \Omega} \max_{j=1, \dots, K} d_j(A) \quad (11.4)$$

The overlapping is called uniform if  $\delta$  is strictly positive. In the present study non-uniform overlapping is also considered. In such cases  $d(A)$  is allowed to vanish at isolated points on the physical boundary (at the corner points of  $\Gamma$ ) remaining positive in the whole interior of  $\Omega$ . The internal distance function  $d_j(A)$  is subsequently used to define the blending functions.

In practice the subdomains are created automatically by the suitable grid generation algorithm. In the first step curvilinear grids are generated which fully surround the physical boundary (in the present case, the inner physical boundary is formed by each airfoil, the shape of outer boundary is arbitrary). In the second step far-field grids are created extending up to 15-50 airfoil chords from the profile. These grids are, as a rule, distant from each airfoil. The remaining void space is filled with rectangular grids. During this procedure the size and the shape of each subdomain must be chosen so that (i) this subdomain does not overlap the profile itself, and (ii) each point of the physical domain is sufficiently overlapped by at least one of the local grids. An example of a grid system generated by this procedure is presented in Fig. 11.1 (the far-field grids are only partially visible).

### 11.3.2 Identification and interpolation procedures

As can be observed in Fig. 11.1 part of each grid boundary may coincide with the physical boundary, while the remaining part forms the interface on which the exchange of information between the subdomains takes place. However, the grid points in the overlap region do not necessarily coincide (e.g. see point A in Fig. 11.1). As a result each interfacial grid point  $C \in \gamma_l$  has to be localized in the reference frame of each grid in the system. This procedure is performed once at the beginning of the calculations; the computational cost is not significant in comparison with the computational cost of a single sweep of the Euler solver.

Once the interfacial points are properly identified the interpolation procedure must be defined in order to prolongate the local solution onto the whole subdomain. In contrast to [9] where high-order interpolation was employed, in the present study bilinear or at most biquadratic interpolation formulas were found to provide satisfactory results.

### 11.3.3 Blending functions.

The second difficulty is related to the fact that the solution obtained on various subdomains remains different in the overlapping regions. This is due to either different discretization or the fact that the solution procedure has reached a different stage in various subdomains.

Despite the above, new boundary condition on the interfaces must be generated using some kind of averaging (e.g. at point B of Fig.11.1).

For this purpose weight functions  $\chi_p$  are defined, (henceforth called *blending functions*), and subsequently are used to construct the global solution. The blending function  $\chi_p$  associated with the  $p$ -th subdomain  $\Omega_p$  is defined as

$$\chi_p(C) := \frac{d_p(C)}{S(C)}, \quad S(C) := \sum_{j=1}^K d_j(C), \quad p = 1, 2, \dots, K. \quad (11.5)$$

where  $d_1, \dots, d_K$  are the *internal distances* described earlier (Eq. 11.3).

The blending functions are non-negative and continuous, forming the partition of unity ( $\chi_1 + \dots + \chi_K \equiv 1$  on  $\Omega$ ). Other properties are described in [9]. One should notice that equations (11.3) and (11.5) have a simple algebraic form and as a result the blending functions can be easily evaluated numerically. Their value can be computed once the overlapping structure is identified at the beginning of all calculations.

### 11.3.4 Global solution and full algorithm

Let us assume that  $U_{[p]}$  is the solution vector defined on the subdomain  $\Omega_p$  and extended to the whole  $\Omega$  by assigning  $U_{[p]}(C) \equiv 0$  for  $C \notin \Omega_p$  (hence  $U_{[p]}$  has discontinuity at  $\gamma_p$ ).

The global solution  $U(C)$  is defined then as

$$U(C) = \sum_{p=1}^K \chi_p(C) U_{[p]}(C), \quad C \in \Omega. \quad (11.6)$$

It should be noted that  $U(C)$  remains continuous on all interfaces despite the fact that each  $U_{[p]}$  is discontinuous on the corresponding interface  $\gamma_p$ . On the other hand, the discontinuities such as shock waves will remain intact in the global solution.

The full algorithm encompasses the following steps:

1. Given the grid in each subdomain  $\Omega_p$ , *localize* interfacial grid points in the cells of remaining grids and evaluate all blending functions on the interfaces.
2. Initialize boundary conditions at the interfaces  $\gamma_p$  ( $p = 1, \dots, K$ ).
3. Perform a few iterations using the Euler solver on each local grid to obtain new local solutions  $U_{[p]}$ , ( $p = 1, \dots, K$ ).
4. Interpolate each local solution on the prescribed interfaces.
5. Evaluate the global function  $U(\cdot)$  on each interface using Eq. (11.6).
6. Check the convergence on all interfaces and if there is no convergence return to step 3.

It is important to point out that the step 5 cannot be performed in parallel. It requires interprocessor communication if different subdomains are served by different processors. However the computational cost associated with the steps 4 and 5 is negligible compared to the cost for step 3. The number of floating point numbers transferred between processors (if different subdomains are served by different processors) is proportional to the number of boundary points, while the cost of step 4 is one order of magnitude higher. As a result one should expect that the benefits of parallelism would increase when increasing grid size.

## 11.4 NUMERICAL RESULTS

The finite volume method presented in Section 2 has been tested in the past for various steady (see e.g. [2, 3, 4]) and unsteady (see e.g. [13]) compressible flows. In this paper, three test cases have been selected to assess the numerical properties of the overlapping mesh-technique in conjunction with the implicit Riemann solver.

The first case is the compressible flow over a 10% circular-arc airfoil. This case was selected for testing the influence of the overlap size on the convergence of the iterative procedure. Various grid systems were used in the numerical experiments with the overlap size  $\delta$  ranging from 1% (case 8b) to 26% (case 8e). The influence of  $\delta$  on the number of iterations required to reach a prescribed accuracy is shown in Fig. 11.2 (details can be found in [7]).

One expects that the number of iterations will increase when  $\delta$  is reduced. However, the present numerical experiments reveal that the influence of the overlap size on the numerical convergence is not significant, though a smaller overlap results in a slightly larger number of iterations. For the transonic case (see [7]) the convergence is affected at low convergence levels, but the resulting number of iterations cannot be fully correlated with the overlap size. It, however, seems that the critical factor may be the proximity of the shock wave to the interface (therefore such configurations should be avoided). The above holds also for cases where more grids are employed.

The second case is the subsonic flow around the GA(W)-1 airfoil with a 29% flap deflected by  $40^\circ$ . The free-stream Mach number is equal to 0.2 and the angle of attack is  $0^\circ$ . Two different grid systems B1 and B2 were chosen here to prove that solution converges when grids are refined (grid system B2 contained roughly twice as many grid points as B1). The total number of grids was 22 with 17 grids actually in contact with one of the airfoil profiles (Fig. 11.1 shows the grid system B1).

The Mach number distribution obtained on the grid system B2 is shown around the flap region in Fig. 11.3. The solid lines in the flowfield correspond to the interfaces of the overlapping grids. The smooth transition of Mach values between the various subdomains can easily be observed in Fig. 11.3.

Quantitatively, the accuracy of the algorithm can be assessed by comparing (Fig. 11.4) the experimental results of [12] with the computed pressure coefficient distributions. The  $C_p$  values are very similarly predicted on both grids and both solutions are in very good agreement with the experiment. In all cases the iterations converge to almost the machine accuracy.

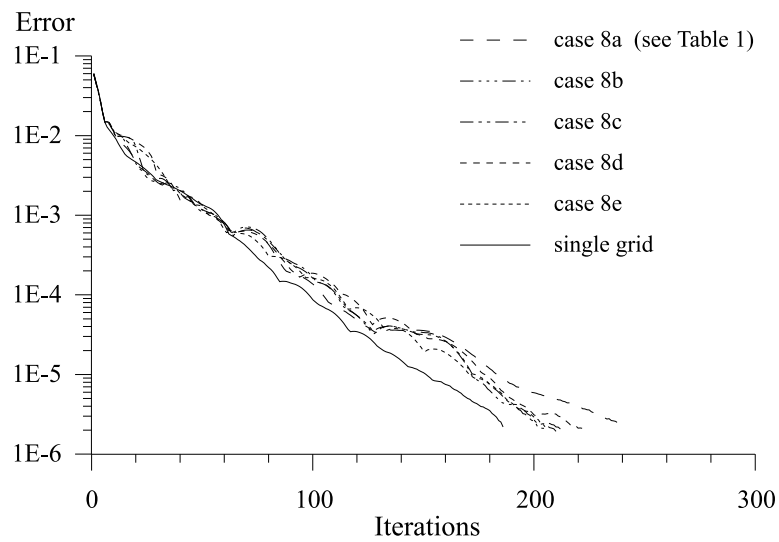


Figure 11.2 Convergence history for various two-grid partitions of the flow domain.

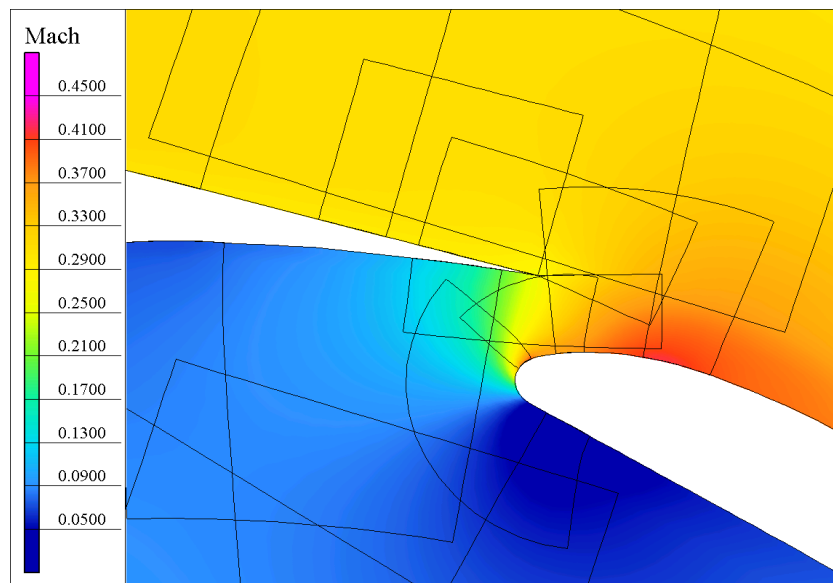


Figure 11.3 Mach number distribution around the GA(W)-1 airfoil for  $M_\infty = 0.2$ , (zoom near the flap); solid lines denote interfaces.

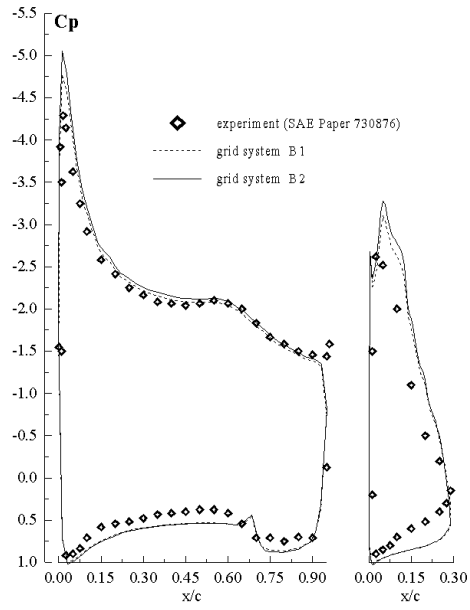


Figure 11.4 Pressure coefficient distribution  $C_p$  on the main airfoil and on the flap of the GA(W)-1 airfoil ( $M_\infty = 0.2$ ).

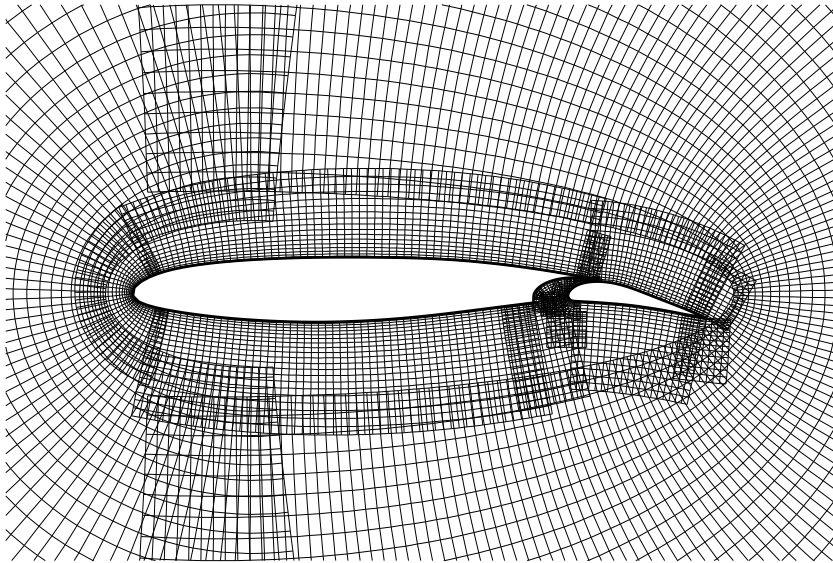


Figure 11.5 Grid system B1 around the SKF-1.1 airfoil.

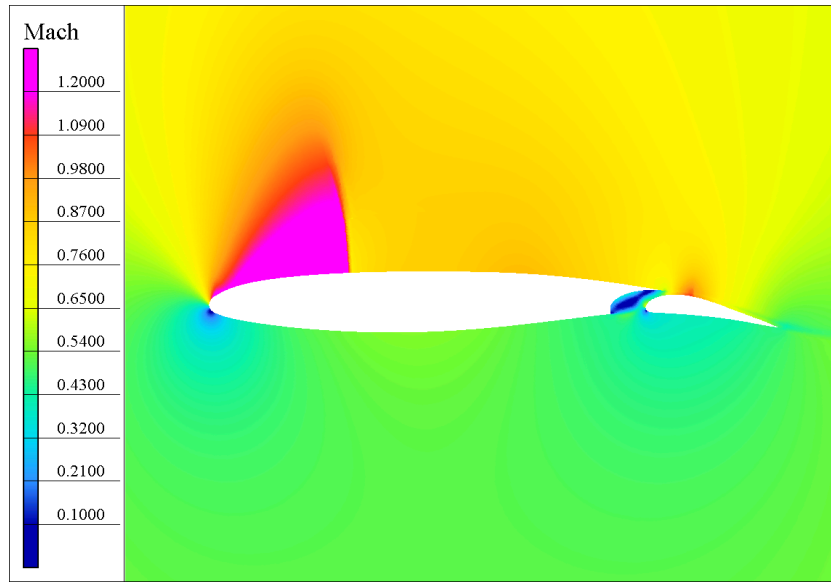


Figure 11.6 Mach number distribution around the SKF-1.1 airfoil for  $M_\infty = 0.60$ ,  $\alpha = 2^\circ$  (case B3).

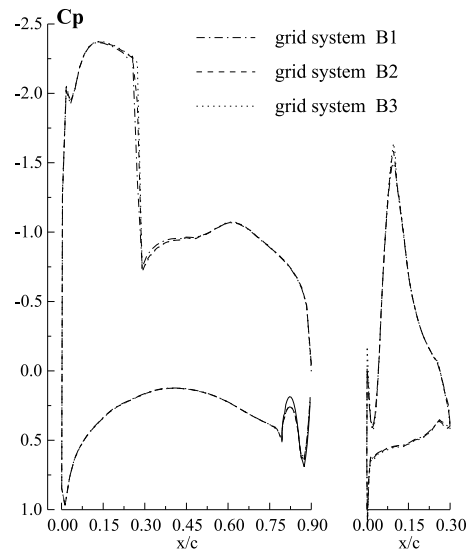


Figure 11.7 Grid convergence study for the SKF-1.1 airfoil for  $M_\infty = 0.60$ ,  $\alpha = 2^\circ$ .



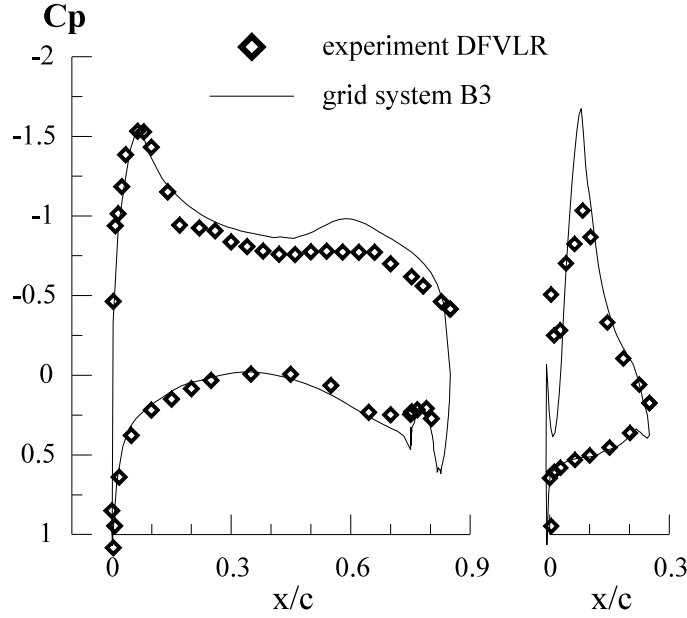


Figure 11.8 SKF-1.1 airfoil - comparison with experiment [10], flow for  $M_\infty = 0.60$ ,  $\alpha = -0.6^\circ$  (case B3).

The third case is the transonic flow around the supercritical SKF-1.1 airfoil with the deflected maneuver-flap [10]. The free-stream Mach number is  $M_\infty = 0.60$  and the angle of attack is  $\alpha = 2^\circ$ . Here again three grid systems B1, B2, and B3 with increasing degrees of refinement, are chosen to demonstrate the convergence of the numerical solution. The systems consist of 20 grids (25 for B3) as shown in Fig. 11.5. The number of grid points is 18410 for B1, 42183 for B2 and 55304 for B3.

The Mach-number field corresponding to the grid system B3 is shown in Fig. 11.6. The shock wave is well visible in the middle part of the main airfoil, while a second, much weaker shock appears over the flap. Figure 11.7 presents the comparison of the computed  $C_p$  distributions on the main airfoil and on the flap for all grid systems used in the computations. Again, the results show that the numerical solutions are grid independent.

Figure 11.8 contains comparisons with the experimental results of [10] for  $M_\infty = 0.60$ ,  $\alpha = -0.6^\circ$ . All in all, a fairly good agreement between computations and experiment has been obtained for the main airfoil profile, but some discrepancies exist on the suction side of the flap.

## 11.5 PARALLEL IMPLEMENTATION

Parallelization of the method is based on the assignment of a single grid or a group of grids to each processor. Therefore, the decomposition of the flowfield into subdomains

determines, at the grid generation stage, the granularity of the parallel problem. As a result, the number of processors  $L$  cannot exceed the total number of grids  $K$ . Moreover, since the grids are of different size, the number of processors  $L$  must be significantly lower than  $K$ , otherwise an uneven load distribution will severely affect the parallel performance.

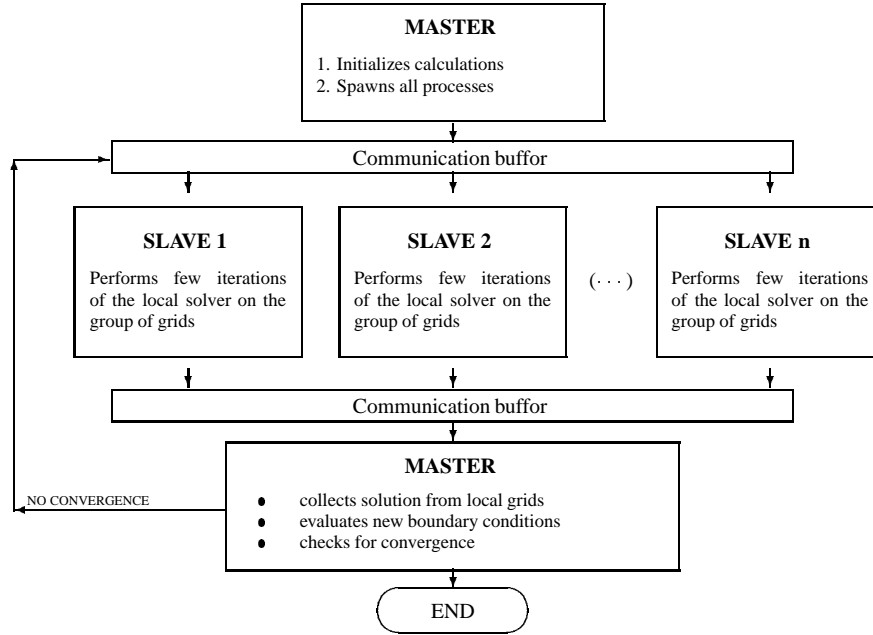


Figure 11.9 Parallel implementation scheme

The parallel implementation was done using the PVM technique (see Fig. 11.9). The *master*-process was responsible for initializing the calculations and controlling all *slave*-processes. Each *slave* was an independent flow solver capable of advancing the iterative process on a single grid or on a prescribed group of grids. After performing a fixed number of iterations, all *slaves* were sending updated boundary data to the *master*-process. The *master*-process was subsequently re-calculating the boundary data, sending them back to the *slaves*. This procedure was repeated until numerical convergence was obtained.

The overall computational effort and total number of iterations does not depend on the number of processors. They were, however, slightly dependent on the number of grids in the system and overlap size, etc. (several results from numerical experiments can be found in [7], see also Fig. 11.2).

The number of inner iterations of the local solver was fixed equal to  $m = 2$ , each of them containing four additional iterations for solving the final linearized system [3, 2]. The increase of  $m$  results in a slight decrease of the number of outer iterations, but the total workload increases.

To estimate the parallel speed-up we have measured the execution time  $\tau_L$  necessary for performing a fixed number of iterations (in our case 100 iterations) on an  $L$ -processor system. The speed-up factor  $\alpha_L$  and the corresponding efficiency  $\eta_L$  were defined as

$$\alpha_L = \frac{\tau_1}{\tau_L}, \quad \eta_L = \frac{\alpha_L}{L} = \frac{\tau_1}{L\tau_L}.$$

### 11.5.1 Static load balancing

Each iteration of the overlapping-mesh algorithm consists of a finite number of fixed workloads associated with each grid. Each workload is practically determined by the number of computational cells in the corresponding grid. This number may vary from grid to grid even by two orders of magnitude (see Table 11.3, where the smallest and largest grids have 24 and 1800 points, respectively). Therefore, grids must be carefully grouped in order to balance the load between processors. This can be done in advance since neither the grids nor their interconnection change during the iteration process.

In order to quantify the load-balancing effects the coefficient  $\beta$  is defined by:

$$\beta_p = \frac{w_p}{\frac{1}{L} \sum_{q=1}^L w_q}, \quad \beta = \max_{p=1, \dots, L} \beta_p,$$

where  $w_p$  stands for the workload assigned to the  $p$ -th processor ( $w_p$  is proportional to the number of grid points assigned to the  $p$ -th processor). The denominator in the first formula describes the workload obtained by the ideal balancing. Therefore, the coefficient  $\beta$  is always larger than 1 and  $\beta = 1$  corresponds to an ideal load-balancing (this analysis is correct for homogeneous systems only). In particular it is evident that the number of processors  $L$  should not significantly exceed

$$L_* := \text{int} \left[ \frac{w_{tot}}{w_*} \right]$$

where  $w_{tot}$  and  $w_*$  stand for the total workload and the workload on the largest grid respectively.

In order to minimize the coefficient  $\beta$ , for a given system of grids, a simple grouping algorithm was used according to which grids (in descending load order) were assigned to processors with the lightest load. The results of this procedure were satisfactory as can be seen in Tables 11.2 and 11.4. Further improvement can be obtained by using a more time consuming algorithm, in which consecutive 2, 3, 4, ... -element permutations of grids (between processors) are considered to minimize the value of the  $\beta$  coefficient.

### 11.5.2 Parallel tests

Various parallel tests were performed for the flows around the GA(W)-1 and SKF 1.1 airfoils using the 16-processor CS6400 computer (for the GA(W)-1 airfoil), and the 64-processor CRAY T3E (for the SKF 1.1 airfoil).

For the GA(W)-1 airfoil the flow domain was covered by a system of 33 overlapping grids. The total workload was 18414 (in terms of grid cells) while the biggest grid had an associated workload of 4000. As a result the maximum number of processors was  $L_* = 5$ . The parallel communication was implemented by using consecutive calls to *pvmfsent* and *pvmfrecv* subroutines.

Table 11.1 presents the acceleration factor  $\alpha_L$ , total efficiency  $\eta_L$  and load balancing coefficient  $\beta$  for computations using up to 5 processors.

Table 11.1 The acceleration factor  $\alpha_L$ , total efficiency  $\eta_L$  and load balancing coefficient  $\beta$  for a 16-processor CS6400 computer (GA(W)-1 profile with 33 overlapping grids).

$L$	$\tau_L$ [sec]	$\alpha_L$	$\eta_L$	$\beta$
1	1185	—	—	—
2	640	1.85	0.93	1.02
3	448	2.65	0.88	1.08
4	367	3.23	0.81	1.04
5	311	3.81	0.76	1.09

Table 11.2 The acceleration factor  $\alpha_L$  total efficiency  $\eta_L$  and load balancing coefficient  $\beta$  for a 64-processor CRAY T3E (SKF-1.1 profile with 25 overlapping grids).

$L$	$\tau_L$ [sec]	$\alpha_L$	$\eta_L$	$\beta$
1	1148	—	—	—
2	585	1.96	0.98	1.0004
3	396	2.90	0.97	1.0005
4	303	3.80	0.95	1.0011
5	245	4.69	0.94	1.028
6	243	4.72	0.78	1.125

For the SKF 1.1 airfoil two different grid systems were used. The first consisted of 25 grids with the total workload of 53344 (grid B3 from previous Section). The largest and the smallest grids had an associated workload of 10000 and 35, respectively, allowing the use of up to 6 processors.

Table 11.3 Workload associated with each of 57 grids for the SKF-1.1 airfoil.

No.	0+	10+	20+	30+	40+	50+
1	504	24	400	440	120	1750
2	360	144	324	72	1350	75
3	468	360	192	42	1350	153
4	112	196	256	48	990	81
5	288	216	1750	64	1800	286
6	70	144	1800	80	1800	195
7	48	360	1350	84	1800	195
8	288	324	1200	1600	1800	—
9	30	224	1225	1600	1200	—
10	144	396	1225	1600	1750	—

Table 11.2 collects the values of  $\alpha_L$ ,  $\eta_L$ ,  $\beta$  obtained on a 64-processor CRAY T3E ( $L = 1, \dots, 6$ ). The case  $L = 6$  is already poorly balanced and, therefore, no further acceleration is possible.

For  $L = 5$  the communication-to-computation performance ratio can be easily observed in Fig. 11.10, which was obtained with the VAMPIR performance analysis tool [6]. Instead of *pvmfsent-pvmfrecv* this implementation used, as a variant, calls to *pvmfreduce-pvmfbcast-pvmfrecv*. No significant performance differences were observed though.

The second grid system contained 57 grids allowing the use of a larger number of processors. Table 11.3 shows the workload  $N_{\xi\eta}$  associated with each grid in the system. The total workload is equal to 36747, and the workload of the largest grid is 1800. Therefore, the maximum number of processors can be up to  $L_* = 20$ . The parallel performance on CRAY T3E is shown in Table 11.4. It is worth noticing that in this case acceleration  $\alpha_L$  grows steadily with growing  $L$  (up to  $L = 19$ ). The drop in efficiency for larger  $L$  can be correlated with a relatively smaller computational load of each processor (only 2000 grid cells are assigned to each processor for  $L = 19$ ).

The variant with calls to *pvmfreduce-pvmfbcast-pvmfrecv* resulted here in improving the acceleration factor by 5-10% (for  $L > 10$ ).

## 11.6 CONCLUSIONS

The present results reveal that the overlapping-mesh technique can be efficiently used on a multi-processor system in conjunction with implicit methods and characteristics-based schemes. Grid-independent solutions were achieved for all the cases considered here. The efficiency of such an approach will increase with the complexity of the problem, provided that the workload can be distributed between the processors as evenly as possible. This can be achieved by further subdivision of existing grids or by re-grouping grids assigned to processors.

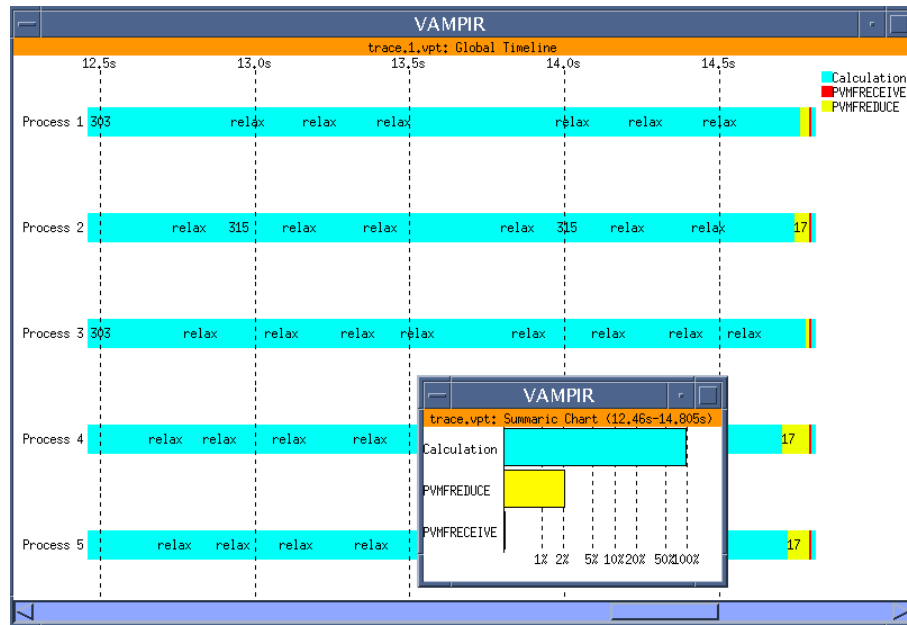


Figure 11.10 Single iteration of the solver for CRAY-T3E using 5 processors - communication to computation performance ratio.

## Acknowledgments

This work has been supported by the KBN 7 T07A 022 14 Grant *The Numerical Simulation of Unsteady Flows in 2-D and 3-D Geometries*. A substantial part of the work was performed when Jacek Rokicki was staying at the Zentrum für Hochleistungsrechnen of TU-Dresden, whose hospitality is kindly acknowledged. Performance tests were run on the CRAY T3E at ZHR, TU-Dresden.

## References

- [1] Benek, J.A., Buning, P.G., Steger, J.L. (1985). A 3-D Chimera Grid Embedding Technique, AIAA Paper 85-1523.
- [2] Drikakis, D., Durst, F. (1994). Investigation of Flux Formulae in Transonic Shock-Wave/Turbulent Boundary Layer Interaction, *Int. Journal for Numerical Methods in Fluids*, **18**, 385-413.
- [3] Drikakis D., Tsangaris S. (1993). On the Accuracy and Efficiency of CFD Methods in Real Gas Hypersonics, *Int. Journal for Numerical Methods in Fluids*, **16**, 759-775.
- [4] Drikakis D., Tsangaris S. (1993). On the Solution of the Compressible Navier-Stokes Equations Using Improved Flux Vector Splitting Methods' *Appl. Math. Modeling*, **17**, 282-297, 1993.

Table 11.4 The acceleration factor  $\alpha_L$ , total efficiency  $\eta_L$  and load balancing coefficient  $\beta$  for a 64-processor CRAY T3E (SKF-1.1 profile with 57 overlapping grids).

$L$	$\tau_L$ [sec]	$\alpha_L$	$\eta_L$	$\beta$
1	770	—	—	—
2	390	1.97	0.99	1.0002
3	264	2.92	0.97	1.0006
4	201	3.83	0.96	1.0004
5	164	4.70	0.94	1.0017
6	139	5.54	0.92	1.003
7	122	6.31	0.90	1.002
8	110	7.00	0.88	1.005
9	100	7.70	0.86	1.006
10	92	8.37	0.84	1.003
11	86.4	8.91	0.81	1.005
12	81.4	9.46	0.79	1.007
13	77.1	9.99	0.76	1.009
14	73.4	10.49	0.75	1.067
15	71.3	10.79	0.72	1.041
16	68.5	11.24	0.70	1.055
17	66.8	11.53	0.68	1.110
18	66.4	11.59	0.64	1.073
19	64.9	11.86	0.62	1.011

- [5] Henshaw, W.D. (1994). A Fourth-Order Accurate Method for the Incompressible Navier-Stokes Equations on Overlapping Grids, *J. Comput. Phys.*, **113**.
- [6] Nagel, W.E., Arnold A., Weber M., Hoppe H-C., Solchenbach K. (1996). VAMPIR: Visualization and Analysis of MPI Resources, *SUPERCOMPUTER* 63, **12**, No. 1, 69-80.
- [7] Rokicki, J., Drikakis, D., Majewski, J., Zóltak, J. (1999). Numerical and Parallel Performance of the Overlapping Mesh Technique Coupled with Implicit Riemann Solver, ZHR Internal Report, ZHR-IR-9901.

- [8] Rokicki, J., Floryan, J.M. (1995). Domain Decomposition and the Compact Fourth-Order Algorithm for the Navier-Stokes Equations, *J. Comput. Phys.*, **116**.
- [9] Rokicki, J., Floryan, J.M. (1999). Unstructured Domain Decomposition Method for the Navier-Stokes Equations, *Computers and Fluids*, **28**, 87-120.
- [10] Stanewsky, E., Thibert, J.J. (1979). Airfoil SKF-1.1 with Maneuver Flap, AGARD Report AR-138.
- [11] Wang, Z.J. (1998). A Conservative Interface Algorithm for Moving Chimera (Overlapped) Grids, *IJCFD*, **10**, 255-265.
- [12] William, H.W. (Jr.) (n.d.). New airfoil Sections for General Aviation Aircraft, SAE Paper 730876.
- [13] Zóltak J., Drikakis D. (1998). Hybrid Upwind Methods for the Simulation of Unsteady Shock-Wave Diffraction Over a Cylinder, *Computer Methods in Applied Mechanics and Engineering*, **162**, No. 1-4, 165-185.



## 12 THE OPTIMIZED ORDER 2 METHOD. APPLICATION TO CONVECTION-DIFFUSION PROBLEMS

Caroline Japhet

*ONERA DTIM/CHP, 29 Av de la Division Leclerc BP 72, 92322 CHATILLON, FRANCE*

japhet@onera.fr

Frédéric Nataf

*CMAP, CNRS URA756, Ecole Polytechnique 91128 Palaiseau Cedex, FRANCE*

nataf@cmapx.polytechnique.fr

Francois Rogier

*Associazione Euratom-ENEA sulla Fusione, ENEA C.R. Frascati  
Via E. Fermi 27, C.P. 65, I-00044 Frascati (Roma) Italy*

rogier@frascati.enea.it

**Abstract:** In fluid dynamics, the convection-diffusion equation models for example the concentration of a pollutant in the air. We present an iterative, non-overlapping domain decomposition method for solving this equation. The domain is divided into subdomains, and the physical problem is solved in each subdomain, with specific conditions at the interfaces. This permits to solve very big problems which couldn't be solved on only one processor. A reformulation of the problem leads to an equivalent problem where the unknowns are on the boundary of the subdomains [14]. The solving of this interface problem by a Krylov type algorithm [15] is done by the solving of independant problems in each subdomain, so it permits to use efficiently parallel computation. In order to have very fast convergence, we use differential interface conditions of order 1 in the normal direction and of order 2 in the tangential direction to the interface, which are optimized approximations of Absorbing Boundary Conditions (ABC) [13], [8]. We present simulations performed on the paragon intel at ONERA (100 processors), which show a very fast convergence, nearly independant both of the physical and the discretization parameters.

## 12.1 THE OPTIMIZED ORDER 2 METHOD

We consider the convection-diffusion problem :

$$\begin{aligned}\mathcal{L}(u) &= cu + a(x, y) \frac{\partial u}{\partial x} + b(x, y) \frac{\partial u}{\partial y} - \nu \Delta u = f \quad \text{in } \Omega \\ \mathcal{C}(u) &= g \quad \text{on } \partial\Omega\end{aligned}\tag{12.1}$$

where  $\Omega$  is a bounded open set of  $\mathcal{R}^2$ ,  $\mathbf{a} = (a, b)$  is the velocity field,  $\nu$  is the viscosity,  $\mathcal{C}$  is a linear operator,  $c$  is a constant which could be  $c = \frac{1}{\Delta t}$  with  $\Delta t$  a time step of a backward-Euler scheme for solving the time dependent convection-diffusion problem, and  $f, g$  are given functions.

The OO2 method is based on an extension of the additive Schwarz algorithm with non-overlapping subdomains :

Let  $\bar{\Omega} = \cup_{i=1}^N \bar{\Omega}_i$ , with  $\Omega_i \cap \Omega_j = \emptyset$ ,  $i \neq j$ . We denote by  $\Gamma_{i,j}$  the common interface to  $\Omega_i$  and  $\Omega_j$ ,  $i \neq j$ . The outward normal from  $\Omega_i$  is  $\mathbf{n}_i$  and  $\boldsymbol{\tau}_i$  is a tangential unit vector. Let  $u$  be the solution of problem (12.1), and  $u_i^p$  the approximation of  $u$  at iteration  $p$  in each subdomain  $\Omega_i$ ,  $1 \leq i \leq N$ . The additive Schwarz algorithm with non-overlapping subdomains is :

$$\begin{aligned}\mathcal{L}(u_i^{p+1}) &= f \quad \text{in } \Omega_i \\ \mathcal{B}_i(u_i^{p+1}) &= \mathcal{B}_i(u_j^p) \quad \text{on } \Gamma_{i,j}, i \neq j \\ \mathcal{C}(u_i^{p+1}) &= g \quad \text{on } \partial\Omega_i \cap \partial\Omega\end{aligned}\tag{12.2}$$

where  $\mathcal{B}_i$  is an interface operator. The original additive Schwarz algorithm [10], with Dirichlet interface conditions ( $\mathcal{B}_i = Id$ ), converge only with overlapping subdomains. In [11], the interface conditions are Robin type conditions ( $\mathcal{B}_i = \frac{\partial}{\partial \mathbf{n}_i} + c_i$ , where  $c_i$  is a

constant), which leads to a convergent algorithm for non-overlapping subdomains. The choice of the interface conditions is fundamental. Many methods has been proposed (see for example [4], [3], [2], [16] ).

The OO2 interface conditions are based on the concept of Absorbing Boundary Conditions (ABC) [5], [6]. This concept enables to understand the mathematical mechanisms on the interfaces, and therefore leads to stable and efficient algorithms.

We introduce first the OO2 interface operator  $\mathcal{B}_i$  based on this concept, and then the substructuring formulation of the method.

### 12.1.1 OO2 interface conditions

It has been proved in [14] that the optimal interface conditions for algorithm (12.1) are the exact Absorbing Boundary Conditions. Unfortunately, as these conditions are not partial differential operators, they are numerically costly and difficult to use. Then, it has been proposed in [13] to use Taylor approximations of order 0 or 2, for low wave numbers, of these optimal interface conditions.

For example, the "Taylor order 0" interface operator is :  $\mathcal{B}_i = \frac{\partial}{\partial \mathbf{n}_i} - \frac{\mathbf{a} \cdot \mathbf{n}_i - \sqrt{(\mathbf{a} \cdot \mathbf{n}_i)^2 + 4c\nu}}{2\nu}$ . (in [4] and [2], the interface conditions can be interpreted as Taylor approximations of order 0).

Numerical tests on a finite difference scheme with overlapping subdomains have shown that the Taylor order 2 interface conditions lead to very fast convergence, compared to the Taylor order 0 or Dirichlet interface conditions, except in the case of a velocity field tangential to the interface, where the convergence is very slow. So, instead of taking low wave numbers approximations, it has been proposed in [7], [8] to use differential interface conditions of order 2 along the interfaces which are "good" approximations of the ABC, not only for low wave numbers, but for a given range of wave numbers. This means that the interface operator is chosen in order to optimize the convergence rate of algorithm (12.1). This "Optimized Order 2" interface operator is defined as follows :

OO2 interface operator :

$$\mathcal{B}_i = \frac{\partial}{\partial \mathbf{n}_i} - \frac{\mathbf{a} \cdot \mathbf{n}_i - \sqrt{(\mathbf{a} \cdot \mathbf{n}_i)^2 + 4c\nu}}{2\nu} + c_2 \frac{\partial}{\partial \tau_i} - c_3 \frac{\partial^2}{\partial \tau_i^2}$$

where  $c_2 = c_2(\mathbf{a} \cdot \mathbf{n}_i, \mathbf{a} \cdot \tau_i)$  and  $c_3 = c_3(\mathbf{a} \cdot \mathbf{n}_i, \mathbf{a} \cdot \tau_i)$  minimize the convergence rate of algorithm (12.1). The analytic analysis in the case of 2 subdomains and constant coefficients in (12.1) reduces the minimization problem to a one parameter minimization problem. This technique is extended in the case of variable coefficients and an arbitrary decomposition, that is only one parameter is computed, and with this parameter we get  $c_2$  and  $c_3$  (see [8]). So the OO2 conditions are easy to use and not costly.

In the case of 2 subdomains, the convergence of algorithm (12.1) with the OO2 interface conditions is proved by computing explicitly the convergence rate  $\rho$  in term of wave numbers  $k$  :

**Theorem 12.1.1** *Let  $\Omega = \mathcal{R}^2$  be decomposed in 2 subdomains  $\Omega_1 = \mathcal{R}^- \times \mathcal{R}$  and  $\Omega_2 = \mathcal{R}^+ \times \mathcal{R}$ . Then,  $\forall \nu > 0$ ,  $c \geq 0$ ,  $a \in \mathcal{R}$ ,  $b \in \mathcal{R}$ ,*

$$\forall k \in \mathcal{R}, \quad |\rho(k, c_2, c_3)| < 1$$

Moreover, when the mesh size tends to 0, the condition number is asymptotically much better for OO2 than for Taylor order 0 or 2 interface conditions :

**Theorem 12.1.2** *Let  $\Omega = \mathcal{R}^2$  be decomposed in 2 subdomains  $\Omega_1 = \mathcal{R}^- \times \mathcal{R}$  and  $\Omega_2 = \mathcal{R}^+ \times \mathcal{R}$ . Let  $a \in \mathcal{R}$ ,  $a \neq 0$ ,  $b = 0$  and  $c \geq 0$  in (12.1). Let  $h$  be the mesh size, and let  $(\rho_{max})_{IC}$  be the maximum of  $\rho$  on  $0 \leq k \leq \frac{\pi}{h}$  with the interface condition IC. Let  $\alpha = 1 + \frac{4\nu c}{a^2}$ . Then, when  $h \rightarrow 0$  :*

$$\begin{aligned} (\rho_{max})_{Taylor \text{ order } 0} &\approx 1 - \frac{2}{\pi} \alpha^{\frac{1}{2}} \left( \frac{|a|h}{\nu} \right) \\ (\rho_{max})_{Taylor \text{ order } 2} &\approx 1 - \frac{4}{\pi} \alpha^{\frac{1}{2}} \left( \frac{|a|h}{\nu} \right) \\ (\rho_{max})_{OO2} &\approx 1 - 8 \alpha^{\frac{1}{8}} \left( \frac{1}{4\pi} \frac{|a|h}{\nu} \right)^{\frac{1}{8}} \end{aligned}$$

An important point is that in the case of  $N$  subdomains (strips) the convergence rate can be estimated in function of the convergence rate of the 2 subdomain case and the decomposition geometry [12]. The convergence is proved by using techniques issued from formal language theory.

## 12.1.2 Substructuring formulation

In [14], the non-overlapping algorithm (12.1) is interpreted as a Jacobi algorithm applied to a problem where the unknowns are on the boundary of the subdomains. That is, the actual unknowns of the problem are the terms  $\mathcal{B}_i(u_i)$  on the interface  $\Gamma_{i,j}$ ,  $i \neq j$ . In the discrete case, this interface problem can be written in the form :

$$D\lambda = b \tag{12.3}$$

where  $\lambda$ , restricted to  $\Omega_i$ , represents the discretization of the term  $\mathcal{B}_i(u_i)$  on the interface  $\Gamma_{i,j}$ ,  $i \neq j$ , and  $D$  is an interface matrix.

To accelerate convergence again, the Jacobi algorithm is replaced by a Krylov algorithm (GMRES, BICG-STAB, ...) [15]. As we use an iterative method, we need only to compute at each step the product  $D\lambda$ , which restriction to  $\Omega_i$  is the discretization of the jump  $\mathcal{B}_i(u_i) - \mathcal{B}_i(u_j)$  on the interface  $\Gamma_{i,j}$ ,  $i \neq j$ , where the  $u_i$ ,  $1 \leq i \leq N$ , are solution of the local problem :

$$\begin{aligned} \mathcal{L}(u_i) &= 0 \quad \text{in } \Omega_i \\ \mathcal{B}_i(u_i) &= \lambda_{i,j} \quad \text{on } \Gamma_{i,j}, \quad i \neq j \\ \mathcal{C}(u_i) &= 0 \quad \text{on } \partial\Omega_i \cap \partial\Omega \end{aligned}$$

with  $\lambda_{i,j}$  the restriction of  $\lambda$  on the interface  $\Gamma_{i,j}$ . So the solving of the interface problem (12.3) by a Krylov method is done by the solving of independant problems in each subdomain, that is we use efficiently parallel computation : each subdomain is assign to a processor which solve his problem independently. The interactions between subdomains are processed by the communications between processors. Once we have computed an approximate solution  $\lambda$  of problem (12.3), we get the approximate solution  $u_i$  of the solution  $u$  of problem (12.1) in each subdomain  $\Omega_i$ ,  $1 \leq i \leq N$ , by solving the local problem :

$$\begin{aligned}\mathcal{L}(u_i) &= f \quad \text{in } \Omega_i \\ \mathcal{B}_i(u_i) &= \lambda_{i,j} \quad \text{on } \Gamma_{i,j}, i \neq j \\ \mathcal{C}(u_i) &= g \quad \text{on } \partial\Omega_i \cap \partial\Omega\end{aligned}$$

## 12.2 NUMERICAL RESULTS

The convection-diffusion equation is discretized by a finite volume scheme [1] (collaboration with Matra BAe Dynamics France). The global domain  $\Omega$  is decomposed in  $N$  non-overlapping subdomains. The interface problem (12.3) is solved by a BICG-STAB algorithm. This involves solving, at each iteration,  $N$  independant subproblems (one per subdomain) which can be performed in parallel.

Each subproblem is solved by a direct method. We denote by  $h$  the mesh size. We compare the results obtained with the OO2 interface conditions and the Taylor order 0 or 2 interface conditions.

*Remark* : The optimized coefficients in the OO2 method are computed in an initialisation step, that is in the computation of the local matrix. They are not computed again in the iterations of BICG-STAB. Moreover, each iteration of BICG-STAB has the same cost for all the interface conditions (Taylor order 0, Taylor order 2, OO2), because the use of order 2 conditions does not increase the bandwidth of the local matrix. So, in the BICG-STAB algorithm, the CPU time is proportional to the number of iterations.

### 12.2.1 Flow in a square

We solve the following problem :

$$\begin{aligned}\mathcal{L}(u) &= 0, \quad 0 \leq x \leq 1, 0 \leq y \leq 1 \\ u(0, y) = 0, \frac{\partial u}{\partial x}(1, y) &= 0, \quad 0 \leq y \leq 1 \\ u(x, 0) = 1, \frac{\partial u}{\partial y}(x, 1) &= 0, \quad 0 \leq x \leq 1\end{aligned} \tag{12.4}$$

We consider a cartesian mesh with constant mesh size  $h$ . The unit square is decomposed in  $N_x \times N_y$  subdomains, where  $N_x$  (resp.  $N_y$ ) is the number of subdomains in the  $x$  (resp.  $y$ ) direction. We consider two types of convection velocity field : a shear velocity

( $a = y$ ,  $b = 0$ ) and a rotating velocity ( $a = -\sin(\pi(y - \frac{1}{2})) \cos(\pi(x - \frac{1}{2}))$ ,  $b = \cos(\pi(y - \frac{1}{2})) \sin(\pi(x - \frac{1}{2}))$ ). The isovalues of the solution of problem (12.3) with the shear velocity are represented in Figure 12.5, and with the rotating velocity in Figure 12.1.

In Table 12.4, we take a decomposition in strips in order to observe the influence on the convergence of the convection velocity angle to the interfaces. We observe that the OO2 interface conditions give a significantly better convergence which is independant of the convection velocity angle to the interfaces. One of the advantages is that for a given number of subdomains, the decomposition of the domain doesn't affect the convergence. Particularly here, for 16 subdomains, the decomposition in strips (Table 12.4) or in squares (Figure 12.2) doesn't affect the convergence.

Figure 12.3 shows that the convergence with the OO2 interface conditions is significantly better for a more general convection velocity (the rotating velocity) and decomposition (in  $4 \times 8$  subdomains).

The convergence with the OO2 interface conditions, for the studied numerical cases, is also nearly independant of the mesh size (see Table 12.1 and Table 12.2). We practically fit to the theoretical estimates of theorem 1.2.

The convergence with the OO2 interface conditions is also very little sensible to the variations of the CFL, as it shown on Table 12.3.

Figure 12.4 shows the  $speed-up = \frac{CPU\ time\ (1\ domain)}{CPU\ time\ (N\ subdomains)}$  of the method. Let  $i_{max}$  (resp.  $j_{max}$ ) be the number of grid points in the  $x$  (resp.  $y$ ) direction, for the global domain. We note  $N_{it}$  the number of BICG-STAB iterations. For a decomposition of the domain in  $N_x \times N_y$  subdomains, the total cost can be estimated by :  $\alpha_1 (\frac{i_{max}}{N_x})^3 \frac{j_{max}}{N_y} + \alpha_2 N_{it} (\frac{i_{max}}{N_x})^2 \frac{j_{max}}{N_y}$ , where  $\alpha_1$  and  $\alpha_2$  are constants. Figure 12.4 shows that for a small number of subdomains, the first term (arising from the LU factorization of the local matrix) is predominant. Then, the second term (arising from the BICG-STAB algorithm) become predominant. After 32 subdomains, the estimate is no more valid, because of the communication costs which can not be neglected.

### 12.2.2 Flow around a cylinder, issued from a Navier-Stokes computation

The convection velocity field is the one of a Navier-Stokes incompressible flow, with Reynolds number  $Re = 10000$ , around a cylinder. This velocity field is issued from a computation performed with the AEROLOG software of the aerodynamic department at Matra BAe Dynamics France. The computational domain is defined by  $\Omega = \{(x, y) = (r \cos(\theta), r \sin(\theta)), 1 \leq r \leq R, 0 \leq \theta \leq 2\pi\}$  with  $R > 0$  given. We solve the following problem :

$$\begin{aligned} \mathcal{L}(u) &= 0 \quad \text{in } \Omega \\ u &= 1 \quad \text{on } \{(x, y) = (\cos(\theta), \sin(\theta)), 0 \leq \theta \leq 2\pi\} \\ u &= 0 \quad \text{on } \{(x, y) = (R \cos(\theta), R \sin(\theta)), 0 \leq \theta \leq 2\pi\} \end{aligned} \quad (12.5)$$

The grid is  $\{(x_i, y_j) = (r_i \cos(\theta_j), r_i \sin(\theta_j)), 1 \leq i, j \leq 65\}$ , and is refined around the cylinder and in the direction of the flow (see Figure 12.5). The isovalues of

Decomposition of the domain	OO2	Taylor order 2	Taylor order 0
normal velocity to the interface $16 \times 1$ subdomains	15	123	141
tangential velocity to the interface $1 \times 16$ subdomains	21	not convergent	86

Table 12.1 Number of iterations versus the convection velocity's angle  $a = y, b = 0, \nu = 1.d - 2, CFL = 1.d9, h = \frac{1}{241}, \log_{10}(Error) < 1.d - 6$

grid	$65 \times 65$	$129 \times 129$	$241 \times 241$
OO2	15	15	15
Taylor order 2	49	69	123
Taylor order 0	49	82	141

Table 12.2 Number of iterations versus the mesh size,  $16 \times 1$  subdomains,  $a = y, b = 0, \nu = 0.01, CFL = 1.d9, \log_{10}(Error) < 1.d - 6$

the solution of problem (12.4) are represented in Figure 12.5 (without the grid) and in Figure 12.6 (with the grid). We note  $N_{max}$  the number of points on the boundary of a subdomain multiply by the number of subdomains. The OO2 interface conditions give also significantly better convergence in that case (Figure 12.7). We observe in Table 12.4 that the convergence is practically independant of the viscosity  $\nu$ .

### 12.3 REMARK

Numerically, the convergence ratio of the method is nearly linear upon the number of subdomains. So it is necessarily to send global information between subdomains, in order to have a convergence ratio independant of the number of subdomains. To tackle this problem, in [9], a "low wave number" preconditioner is applied to the OO2 method.

### 12.4 CONCLUSION

The OO2 method applied to convection-diffusion problems appears to be a very efficient method. Its main advantage is that it is a general domain decomposition technique, with no a priori knowledge of the boundary layers or the recirculation zones location. The convergence ratio is numerically nearly independant both of the physical parameters and the discretization parameters.

grid	$65 \times 65$	$129 \times 129$	$241 \times 241$
OO2	25	26	30
Taylor order 0	76	130	224

Table 12.3 Number of iterations versus the mesh size,  $4 \times 4$  subdomains, rotating velocity,  $a = -\sin(\pi(y - \frac{1}{2}))\cos(\pi(x - \frac{1}{2}))$ ,  $b = \cos(\pi(y - \frac{1}{2}))\sin(\pi(x - \frac{1}{2}))$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $\log_{10}(Error) < 1.d - 6$

	$CFL = 1.d0$	$CFL = 1.d3$	$CFL = 1.d5$	$CFL = 1.d9$
OO2	3	12	15	15
Taylor ordre 2	2	21	58	123
Taylor ordre 0	3	18	48	141

Table 12.4 Number of iterations versus the CFL,  $16 \times 1$  subdomains,  $a = y$ ,  $b = 0$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$ ,  $\log_{10}(Error) < 1.d - 6$

	OO2	Taylor order 2	Taylor order 0
$\nu = 1.d - 5$	56	41	119
$\nu = 1.d - 4$	43	121	374
$\nu = 1.d - 3$	32	$N_{max} = 768$ $\log_{10}(Error) = -5.52$	$N_{max} = 768$ $\log_{10}(Error) = -2.44$

Table 12.5 Number of iterations versus the viscosity,  $4 \times 2$  subdomains, Navier-Stokes flow velocity,  $\nu = 1.d - 4$ ,  $CFL = 1.d9$ ,  $\log_{10}(Error) < 1.d - 6$



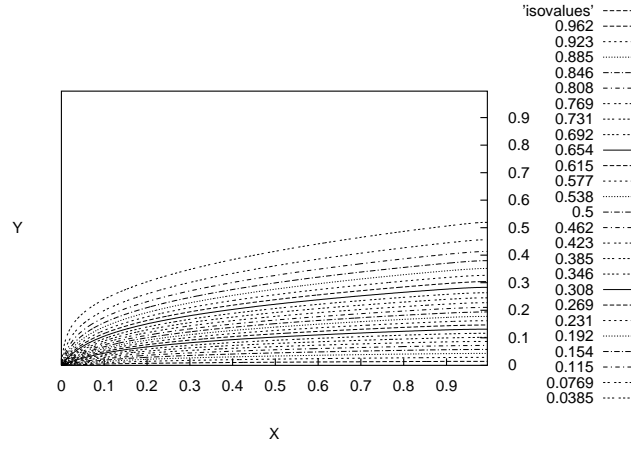


Figure 12.1 Isovalues of the solution  $u$ , shear velocity,  $a = y$ ,  $b = 0$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$

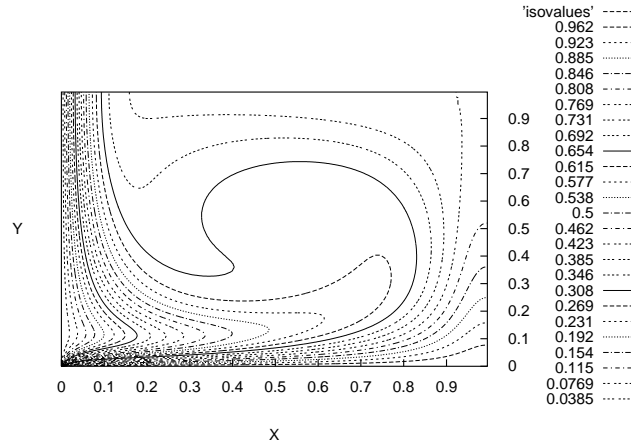


Figure 12.2 Isovalues of the solution  $u$ , rotating velocity,  $a = -\sin(\pi(y - \frac{1}{2}))\cos(\pi(x - \frac{1}{2}))$ ,  $b = \cos(\pi(y - \frac{1}{2}))\sin(\pi(x - \frac{1}{2}))$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$

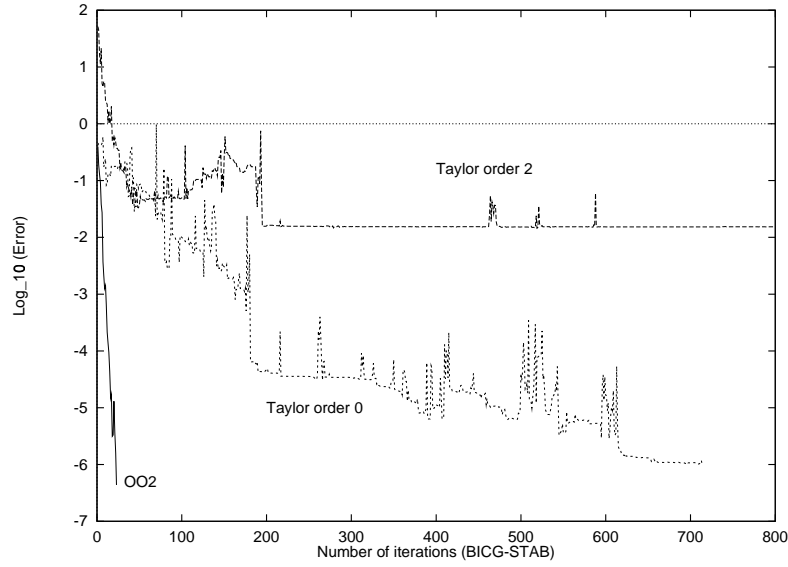


Figure 12.3 Error versus the number of iterations,  $4 \times 4$  subdomains, shear velocity,  $a = y$ ,  $b = 0$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$

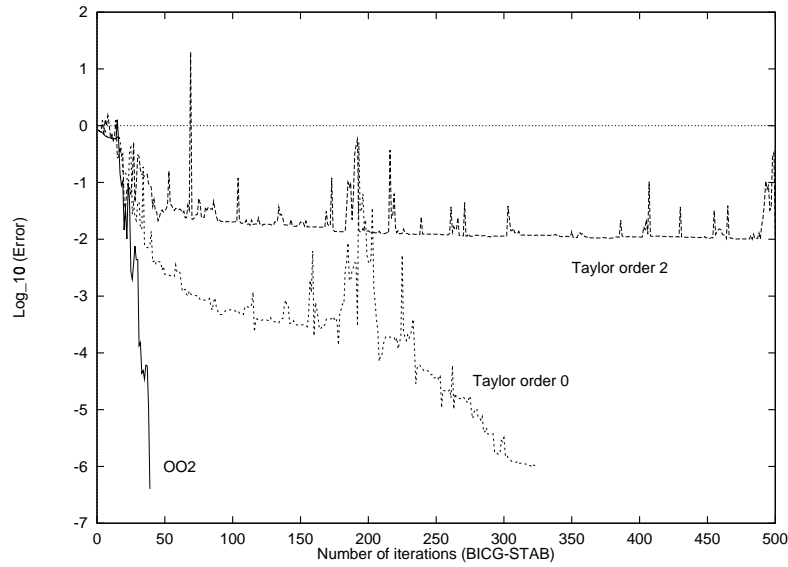


Figure 12.4 Error versus the number of iterations,  $4 \times 8$  subdomains, rotating velocity,  $a = -\sin(\pi(y - \frac{1}{2})) \cos(\pi(x - \frac{1}{2}))$ ,  $b = \cos(\pi(y - \frac{1}{2})) \sin(\pi(x - \frac{1}{2}))$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$

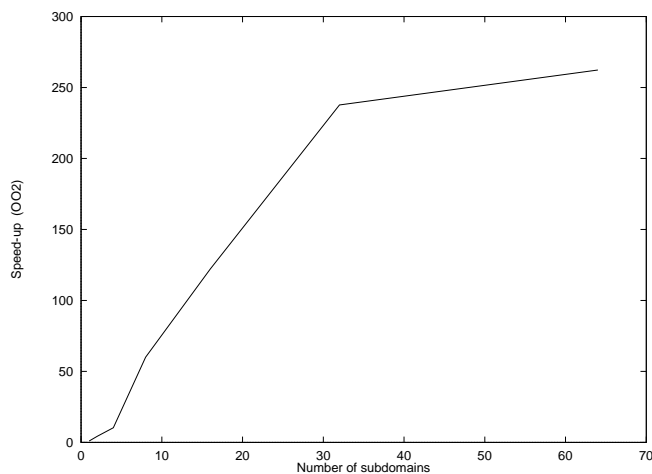


Figure 12.5 Speed-up versus the number of subdomains, for decompositions in  $2 \times 1$ ,  $2 \times 2$ ,  $4 \times 2$ ,  $4 \times 4$ ,  $8 \times 4$  and  $8 \times 8$  subdomains, shear velocity,  $a = y$ ,  $b = 0$ ,  $\nu = 1.d - 2$ ,  $CFL = 1.d9$ ,  $h = \frac{1}{241}$ ,  $Residual < 1.d - 9$

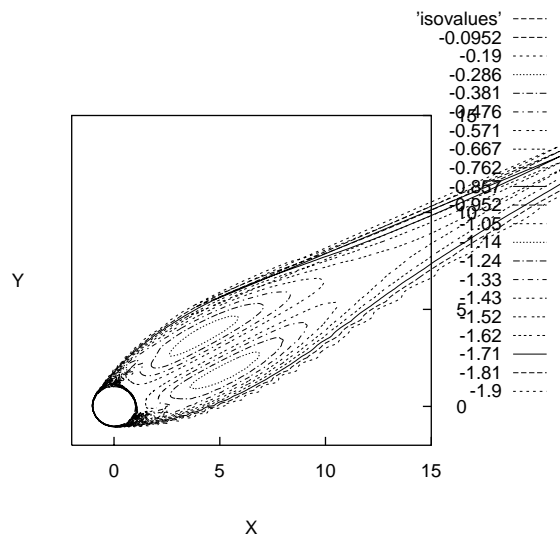


Figure 12.6 Isovalues of the solution  $u$ , Navier-Stokes flow velocity,  $\nu = 1.d - 4$ ,  $CFL = 1.d9$

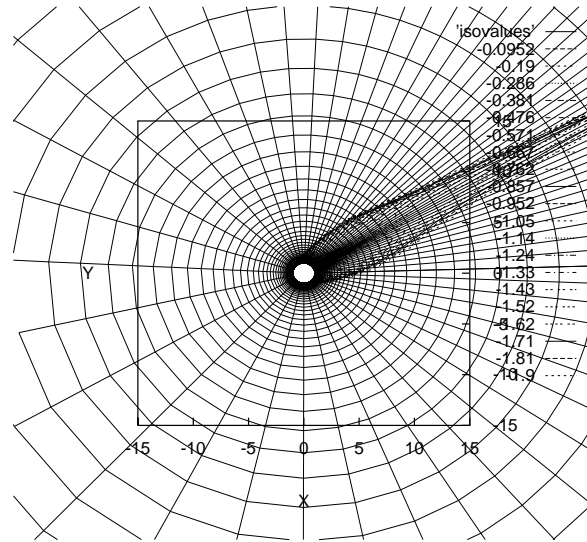


Figure 12.7 Grid and isovalues of the solution  $u$ , Navier-Stokes flow velocity,  $\nu = 1.d - 4$ ,  $CFL = 1.d9$

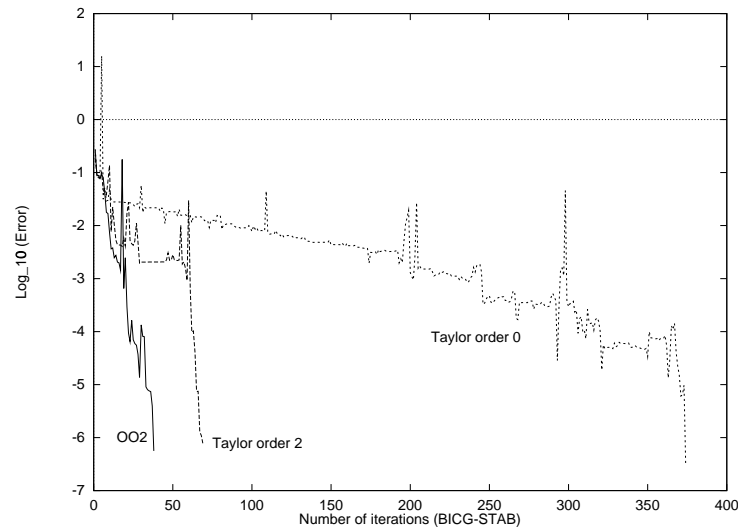


Figure 12.8 Error versus the number of iterations,  $4 \times 2$  subdomains, Navier-Stokes flow velocity,  $\nu = 1.d - 4$ ,  $CFL = 1.d9$

## References

- [1] C. Borel and M. Bredif. High Performance Parallelized Implicit Euler Solver For The Analysis Of Unsteady Aerodynamic Flows. *First European CFD Conference, Brussels*, september 1992.
- [2] C. Carlenzoli and A. Quarteroni. Adaptive domain decomposition methods for advection-diffusion problems. In I. Babuska and al., editors, *Modeling, Mesh Generation and Adaptive Numerical Methods for Partial Differential Equations*, The IMA Volumes in Mathematics and its Applications, 75, pages 165-187. Springer Verlag, 1995.
- [3] P. Charton, F. Nataf, F. Rogier. Méthode de décomposition de domaine pour l'équation d'advection-diffusion. *C. R. Acad. Sci. Paris*, 313, Série I, pages 623-626, 1991.
- [4] B. Desprès. Décomposition de domaine et problème de Helmholtz. *C.R. Acad. Sci., Paris*, 311, Série I, pages 313-316, 1990.
- [5] B. Engquist and A. Majda. Absorbing Boundary Conditions for the Numerical Simulation of Waves. *Math. Comp.*, 31 (139), pages 629-651, 1977.
- [6] L. Halpern. Artificial Boundary Conditions for the Advection-Diffusion Equations. *Math. Comp.*, 174, pages 425-438, 1986.
- [7] C. Japhet. Optimized Krylov-Ventcell Method. Application to Convection-Diffusion Problems. *9<sup>th</sup> International Conference on Domain Decomposition Methods, Bergen (Norway), Domain Decomposition Methods in Sciences and Engineering, edited by P. Bjorstad, M. Espedal and D. Keye*, p. 382-389, 1997.
- [8] C. Japhet. Méthode de décomposition de domaine et conditions aux limites artificielles en mécanique des fluides: Méthode Optimisée d'Ordre 2. Thèse de doctorat, Université Paris XIII, 1998.
- [9] C. Japhet, F. Nataf and F.X. Roux. The Optimized Order 2 Method with a coarse grid preconditioner. Application to Convection-Diffusion Problems. *10<sup>th</sup> International Conference on Domain Decomposition Methods, Boulder (USA), Contemporary Mathematics*, series AMS, Vol. 218, pp. 279-286, 1998.
- [10] P.L. Lions. On the Schwarz Alternating Method I. In *First Int. Symp. on Domain Decomposition Methods (R. Glowinski, G.H. Golub, G.A. Meurant and J. Periaux, eds)*, SIAM (Philadelphia, PA), 1988.
- [11] P.L. Lions. On the Schwarz Alternating Method III: A variant for Nonoverlapping Subdomains. In *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM, pages 202-223, 1989.
- [12] F. Nataf and F. Nier. Convergence rate of some domain decomposition methods for overlapping and nonoverlapping subdomains. *Numerische Mathematik*, 75, pages 357-377, 1997.
- [13] F. Nataf and F. Rogier. Factorisation of the Convection-Diffusion Operator and the Schwarz Algorithm. *M<sup>3</sup>AS*, 5 (1), pages 67-93, 1995.

- [14] F. Nataf, F. Rogier and E. de Sturler. Domain decomposition methods for fluid dynamics. In A. Sequeira, editor, *Navier-Stokes equations on related non linear analysis*, pages 307-377. Plenum Press Corporation, 1995.
- [15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.
- [16] K.H. Tan and M.J.A. Borsboom. On Generalized Schwarz Coupling Applied to Advection-Dominated Problems. *Contemporary Mathematics*, 180, pages 125-130, 1994.

# 13    A PARALLEL ALGORITHM FOR AN ELASTOHYDRODYNAMIC PIEZOVISCOUS LUBRICATION PROBLEM

M. Arenaz, R. Doallo and J. Touriño

*Department of Electronics and Systems  
University of A Coruña  
Spain*

arenaz@des.fi.udc.es, doallo@udc.es and juan@udc.es

C. Vázquez

*Department of Mathematics  
University of A Coruña  
Spain*

carlosv@udc.es

**Abstract:** This work presents a parallel version of a complex numerical algorithm to solve a lubrication problem studied in Tribology. The execution of the sequential algorithm on a workstation requires a huge amount of CPU time and memory resources. So, in order to reduce the computational cost, we have applied parallelization techniques to the most costly parts of the original source code. Some blocks of the sequential code were also redesigned for the execution on a multiprocessor. In this paper, we describe our parallel version, we show some running time measures that illustrate its efficiency, and we present some numerical results that represent the solution of our lubrication problem.

## Introduction

Our work is concerned with the analysis of the lubricant behaviour of an industrial device that arises in Mechanical Engineering. For a wide range of these devices, such as the journal bearing device, the main task is to calculate, for a given imposed load, the pressure distribution of the lubricant fluid and the gap between two surfaces in contact [2].

Most of the lubricated devices appearing in industrial applications can be represented by means of a ball-bearing geometry (Figure 13.1). In this geometry, the industrial device surfaces are represented by a sphere and a plane. In order to perform a realistic numerical simulation of the device, the problem is described by a mathematical model (Section 13.1) of the displacement of the fluid between a rigid plane and an elastic and loaded sphere. An approximation to the solution of the mathematical model is calculated using a numerical algorithm (Section 13.2) that includes fixed point and finite element techniques, and duality methods (see [4] for further details).

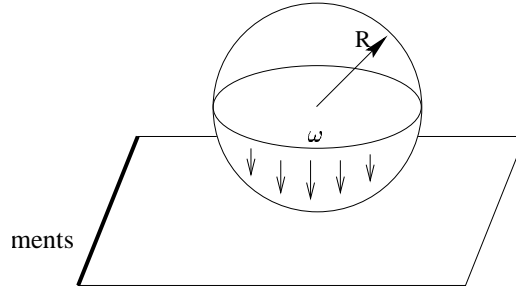


Figure 13.1 Ball-bearing geometry.

In this work, we focus on the parallelization of the algorithm (Section 13.3) on the Fujitsu AP3000 multiprocessor using MPI as message-passing library. We also present running time measures that illustrate the efficiency of the parallel code, and some numerical results corresponding to an approximation of the solution of the lubrication problem (Section 13.4). Finally, conclusions and future work are discussed (Section 13.5).

### 13.1 DESCRIPTION OF THE PROBLEM

In order to perform a realistic numerical simulation of the device, an appropriate mathematical model must be considered. Our model is posed over a two-dimensional domain (Figure 13.2) which represents the mean plane of contact between the two surfaces in contact in the ball-bearing geometry (Figure 13.1).

In our mathematical formulation, the goal is to determine the pressure  $p$  and the saturation  $\theta$  of the lubricant, the gap  $h$  between the ball and the plane, and the



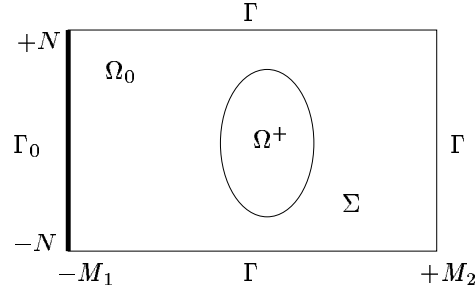


Figure 13.2 Ball-bearing two-dimensional domain  $\Omega$ .

minimum reference gap  $h_0$ . The model consists of the following set of nonlinear partial differential equations:

$$\frac{\partial}{\partial x} \left( e^{-\alpha p} h^3 \frac{\partial p}{\partial x} \right) + \frac{\partial}{\partial y} \left( e^{-\alpha p} h^3 \frac{\partial p}{\partial y} \right) = 12s\nu \frac{\partial}{\partial x} h, \quad p > 0, \quad \theta = 1 \text{ in } \Omega^+ \quad (13.1)$$

$$\frac{\partial}{\partial x} (\theta h) = 0, \quad p = 0, \quad 0 \leq \theta \leq 1 \text{ in } \Omega_0 \quad (13.2)$$

$$e^{-\alpha p} h^3 \frac{\partial p}{\partial \vec{n}} = 12s\nu(1 - \theta) h \cos(\vec{n}, \vec{i}), \quad p = 0 \text{ in } \Sigma \quad (13.3)$$

$$h = h(x, y, p) = h_0 + \frac{x^2 + y^2}{2R} + \frac{2}{\pi E} \int_{\Omega} \frac{p(t, u)}{\sqrt{(x - t)^2 + (y - u)^2}} dt du \quad (13.4)$$

$$\theta = \theta_0 \text{ on } \Gamma_0 \quad (13.5)$$

$$p = 0 \text{ on } \Gamma \quad (13.6)$$

$$\omega = \int_{\Omega} p(x, y) dx dy \quad (13.7)$$

where the unknown vector is  $(p, \theta, h, h_0)$  and the two-dimensional domain, the lubricated region, the cavitated region, the free boundary, the supply boundary and the boundary at atmospheric pressure are, respectively (see Figure 13.2):

$$\begin{aligned} \Omega &= (-M_1, M_2) \times (-N, N) \\ \Omega^+ &= \{(x, y) \in \Omega / p(x, y) > 0\} \\ \Omega_0 &= \{(x, y) \in \Omega / p(x, y) = 0\} \\ \Sigma &= \partial\Omega^+ \cap \Omega \\ \Gamma_0 &= \{(x, y) \in \partial\Omega / x = -M_1\} \\ \Gamma &= \partial\Omega \setminus \Gamma_0 \end{aligned}$$

being  $M_1$ ,  $M_2$  and  $N$  positive constants. The input data are shown in Table 13.1. These are the velocity field  $(s, 0)$ , the piezoviscosity coefficients  $\nu_0$  and  $\alpha$ , the unit vector  $\vec{n}$  normal to  $\Sigma$  pointing to  $\Omega_0$ , the unit vector in the  $x$ -direction  $\vec{i}$ , the Young equivalent modulus  $E$ , the sphere radius  $R$ , and the load  $\omega$  imposed on the device in a normal to the plane direction. Equations 13.1-13.3 model the lubricant pressure behaviour, assuming the Barus pressure-viscosity relation. Equation 13.4 establishes the relation between the gap and the lubricant pressure. Equation 13.7 balances the hydrodynamic and the external loads. Consult [3] for a wider explanation of the physical motivation and the mathematical modelling.

Table 13.1 Input data of the numerical algorithm.

Symbol	Physical parameter
$(s, 0)$	Velocity field
$\nu_0$	Piezoviscosity coefficient
$\alpha$	Piezoviscosity coefficient
$\vec{n}$	Normal unit vector
$\vec{i}$	$X$ -axis unit vector
$h_0$	Minimum reference gap
$E$	Young equivalent modulus
$R$	Sphere radius
$\omega$	Imposed load

## 13.2 THE NUMERICAL ALGORITHM

The numerical algorithm corresponding to the mathematical model mainly combines fixed point techniques, finite elements and duality methods [4]. The structure of the numerical algorithm is depicted in Figure 13.3.

In order to obtain a more accurate approach of different real magnitudes, it is interesting to handle finer meshes. This mesh refinement involves a great increase in storage cost and execution time. So, the use of high performance computing techniques is required in order to reduce the impact of this computational cost problem.

As we can see in Figure 13.3, the algorithm consists of four nested loops called load, gap, characteristics and multipliers loops, respectively. The execution begins using a predefined value of the gap parameter  $h_0$ . In the two innermost loops, namely characteristics and multipliers, the pressure  $p$  and the saturation  $\theta$  of the lubricant fluid are computed (Blocks 5 and 8, respectively). The computations in these two loops mainly involve the recursive finite element solution of linearized partial differential equations, obtained from the application of a transport-diffusion technique combined with a multiplier method for the saturation variable. In gap loop, the gap  $h$  between the sphere and the plane is updated (Block 10) in terms of the pressure following

```

PROGRAM tribology
  load_loop:      DO i = 1, maxp
    gap_loop:      DO j = 1, maxq
      Compute finite element matrix C          ! *** Block 1 ***
      Compute second member fixed in characteristics ! *** Block 2 ***
    characteristics_loop: DO k = 1, maxr
      Compute second member fixed in multipliers ! *** Block 3 ***
    multipliers_loop: DO l = 1, maxs
      Compute final second member b          ! *** Block 4 ***
      Linear system solution  $Cp = b$           ! *** Block 5 ***
      Compute multiplier                      ! *** Block 6 ***
      IF (multiplier_convergence) EXIT          ! *** Block 7 ***
    END DO multipliers_loop
      Compute fluid saturation  $\theta$           ! *** Block 8 ***
      IF (pressure_convergence) EXIT          ! *** Block 9 ***
    END DO characteristics_loop
      IF (k == maxr + 1) EXIT load_loop
      Update gap h                          ! *** Block 10 ***
      IF (gap_convergence) EXIT              ! *** Block 11 ***
    END DO gap_loop
      IF (j == maxq + 1) EXIT load_loop
      Compute hydrodynamic load              ! *** Block 12 ***
      IF (load_convergence) EXIT              ! *** Block 13 ***
    END DO load_loop
  END PROGRAM tribology

```

Figure 13.3 Pseudocode of the numerical algorithm in Fortran90-style. The parameters  $maxp$ ,  $maxq$ ,  $maxr$  and  $maxs$  represent the maximum number of iterations for load, gap, characteristics and multipliers loops, respectively.

Equation 13.4. This costly process requires the integration over the whole domain for each finite element node. Finally, the hydrodynamic load generated by the pressure of the fluid is computed (Block 12) by means of numerical integration in Equation 13.7. If the computed integral is close to the imposed load, the algorithm finishes. Otherwise, a new value  $h_0$  is determined by a bisection technique and the algorithm performs a new iteration in load loop. For a wider explanation of this complex algorithm we address the reader to [4].

### 13.3 THE PARALLELIZATION PROCESS

In this section, we present a version of the algorithm for a distributed-memory multiprocessor. We have focussed on the most costly parts of the algorithm in terms of execution time. So, firstly, we have performed an analysis of the distribution of the execution time, which led us to the conclusion that approximately 96% of this time concentrates on multipliers loop and on the *update gap h* functional block (see

Figure 13.3). In spite of that, as we will see later in this section, it is necessary to parallelize more parts of the algorithm. Hereafter, we will describe the parallelization process of the most interesting blocks.

First, in the *computation of the final second member  $b$*  (Block 4 in Figure 13.3), finite element discretization allows to obtain it in two different ways: by traversing the set of finite elements or by traversing the set of nodes of the mesh. The difference between both methods lies in the order in which the contributions of all finite elements to the final value of  $b$  are summed up. In the first method, implemented in the original sequential code, vector  $b$  is computed as follows:

$$\forall \text{ triangular finite element } \phi_k \\ b[\gamma(\phi_k, i)] = \sum \text{Contrib}(\phi_k, i) \quad , \quad i = 1, 2, 3$$

where  $\gamma(\phi_k, i)$  represents the number of the node of the mesh that is associated with the  $i^{th}$  vertex of the element  $\phi_k$ , and  $\text{Contrib}(\phi_k, i)$  is the contribution of the element  $\phi_k$  to the node numbered as  $\gamma(\phi_k, i)$ . Note that vector  $b$  is referenced through an indirect index. The parallelization of this algorithm introduces communications overhead that reduces the efficiency of the parallel program.

The second method overrides the problem described in the previous paragraph by using the following algorithm:

$$\forall \text{ node } j \text{ of the mesh} \\ b[j] = \sum \text{Contrib}(\phi_k, i) \quad \forall \text{ triangular finite element } \phi_k \quad , \quad i = 1, 2, 3$$

The advantage of this method lies in the fact that it is cross-iteration independent, that is, the calculation of the value associated to any node does not depend on the value corresponding to any other node. Therefore, computations can be distributed in the most appropriate way for the subsequent stages of our program.

In order to *solve the symmetric positive definite system  $Cp = b$*  (Block 5 in Figure 13.3) at each step of the innermost loop, we have experimented with a parallel sparse conjugate gradient method [1, Chap.2], and with a parallel sparse Cholesky method [6], combined with reordering strategies to reduce the fill-in in matrix  $C$  [5, Chap.8]. Although we have checked that both methods are highly scalable for the characteristics of our algorithm, we have chosen the second one because it was proved to be less costly in terms of execution time. Note that  $b$  varies at each step of multipliers loop, and that the finite element matrix  $C$  is the same in each iteration of gap loop (see Figure 13.3).

The *computation of the multiplier* (Block 6 in Figure 13.3) consists of a loop where there are true dependences [8] among the values of the multiplier on the last nodes (those belonging to the right boundary of the domain  $\Omega$ ). This block can be parallelized efficiently by performing the calculations corresponding to the last nodes on the same processor. So, we have used a standard block distribution satisfying this constraint.

At this point, multipliers loop is parallelized but characteristics loop is executed in a sequential manner, which makes necessary to perform two *gather* operations for each iteration in the latter loop. This communications overhead was reduced by parallelizing the blocks included in characteristics loop as follows. First, the *computation of the second member fixed in multipliers* (Block 3 in Figure 13.3) has been implemented, on the one hand, by using a cross-iteration independent algorithm that traverses the set of nodes of the mesh and, on the other hand, by imposing a constraint on the data distribution so that the computations corresponding to the nodes located at the supply boundary  $\Gamma_0$  are assigned to the same processor. Second, the *computation of the fluid saturation* (Block 8 in Figure 13.3) and the *convergence test on pressure* (Block 9 in Figure 13.3) are also cross-iteration independent algorithms, so their computations have been distributed according to the standard block distribution subject to the two previously described constraints.

Regarding the *updating of the gap* (Block 10 in Figure 13.3), it is one of the most costly stages of the original sequential algorithm. For each node of the mesh, the gap is calculated according to Equation 13.4. This computation is cross-iteration independent, which makes the efficiency of the parallelization process independent of the data distribution scheme. So, in order to avoid data redistributions, we have used the same distribution as in the already described functional blocks.

## 13.4 EXPERIMENTAL RESULTS

In this section, we present results in terms of execution times (Table 13.2) for different architectures, as well as approximated numerical results (Figures 13.4-13.7) for the solution of our lubrication problem, that is, for the pressure, the saturation and the gap. In our tests, we have used a uniform mesh composed of 38400 triangular elements and 19521 nodes to discretize the domain of the problem  $\Omega$ . The nodes are ordered up and down, left and right so that the first nodes correspond to the boundary  $\Gamma_0$ . The dimensions of the sparse matrix  $C$  are  $19521 \times 19521$ , and the number of non-zero entries is 96961.

In a previous work [7], we described an efficient version of the algorithm specifically developed for the Fujitsu VP2400 vector computer. This version proved to be very suitable for a vector architecture, as demonstrates the fact that the percentage of vectorization of the source code was approximately 96%. The running time of the whole program using the mesh described above was  $43h : 38m : 38s$ , from which  $33h : 45m : 50s$  were executed on the vector unit of the VP2400.

In order to reduce the computational cost of the algorithm, we developed a parallel version for the Fujitsu AP3000 distributed-memory multiprocessor. This machine consists of UltraSparc-II microprocessors at 300MHz interconnected in a two-dimensional torus topology. The parallel program was written in Fortran77 using MPI as message-passing library. Table 13.2 shows execution time measures for the most costly parts of the algorithm, and for up to eight processors (this limit was imposed by the configuration of our parallel machine). The last two rows make reference to the running time of the whole program and to the corresponding speedups, respectively. The results

Table 13.2 Execution times and speedups. Times are displayed in hours, minutes and seconds (*hh : mm : ss*).

Number of PEs	1	2	4	8
Characteristics loop	10:02	6:45	4:10	2:58
Multipliers loop	124:38:18	71:25:16	38:01:34	23:34:51
Update gap h block	8:33:59	4:36:08	2:21:15	1:12:47
Whole algorithm	133:23:53	76:09:04	40:27:48	24:51:18
Speedup	1.00	1.75	3.30	5.37

for one processor are estimations based on the execution times of one iteration of characteristics, multipliers and gap loops. The total number of iterations in multipliers loop was 1640221. We disposed of coarser meshes, but it is not worth executing those tests on a multiprocessor because their computational cost is not high enough.

The reduction in the execution time of the program is significant. As we can see, good speedups were achieved by restructuring the sequential code and applying an adequate data distribution to reduce the communications overhead that arises from the described dependences.

The aim of this work was not only to reduce computing time, but also to test in practice the convergence of the finite element space discretization. Next, we present some numerical results corresponding to the uniform mesh described at the beginning of this section (38400 triangular elements and 19521 nodes). The real parameter that is relevant in our tests is the load  $\omega$  that is imposed on the industrial device. In Figures 13.4 and 13.6, approximation profiles for the pressure  $p$  (set of curves with a maximum) and the gap  $h$  (set of curves with a minimum) are depicted for  $\omega = 3$  and  $\omega = 5$ , respectively. In Figures 13.5 and 13.7, the saturation  $\theta$  profile is shown, respectively, for the same values of  $\omega$ . In the figures, the  $x$ -axis represents the longitudinal section of the domain. Each curve represents the approximated values of a physical magnitude ( $p$ ,  $h$  or  $\theta$ ) in the nodes of the mesh that have the same  $y$ -coordinate, being the highest curve the one corresponding to the central line.

Although the analysis of the numerical results is beyond the scope of this paper, we briefly comment the most outstanding aspects of the graphs presented before. In Figure 13.4, we can see that the sphere and the plane are more close to each other around the contact point. For this reason, in that region, the gap curves have their minimum while the pressure ones have their maximum. Besides, there is no cavitation where the pressure is positive. Therefore, the saturation is equal to the unit (see Figure 13.5).

### 13.5 CONCLUSIONS AND FUTURE WORK

In this work, we have described the parallelization of a numerical code to solve an elastohydrodynamic piezoviscous lubrication problem. The parallel algorithm reduces

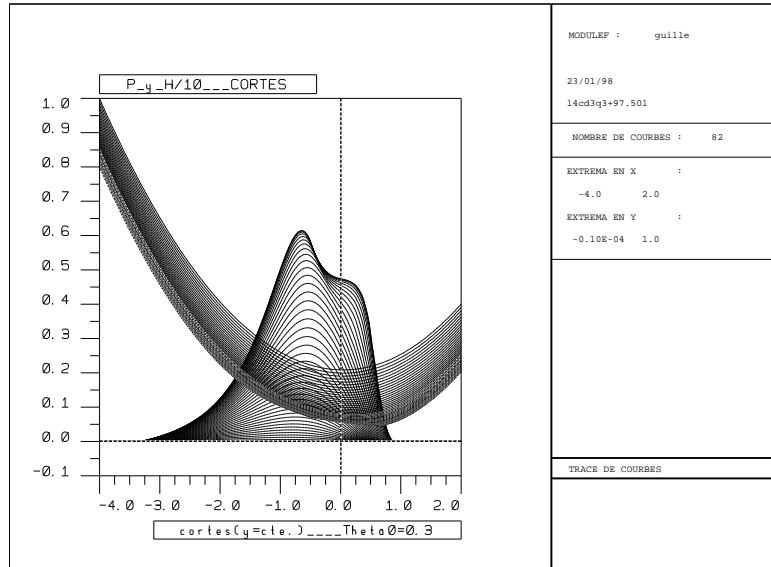


Figure 13.4 Pressure and gap profiles for  $\omega = 3$  and  $\theta_0 = 0.3$ .

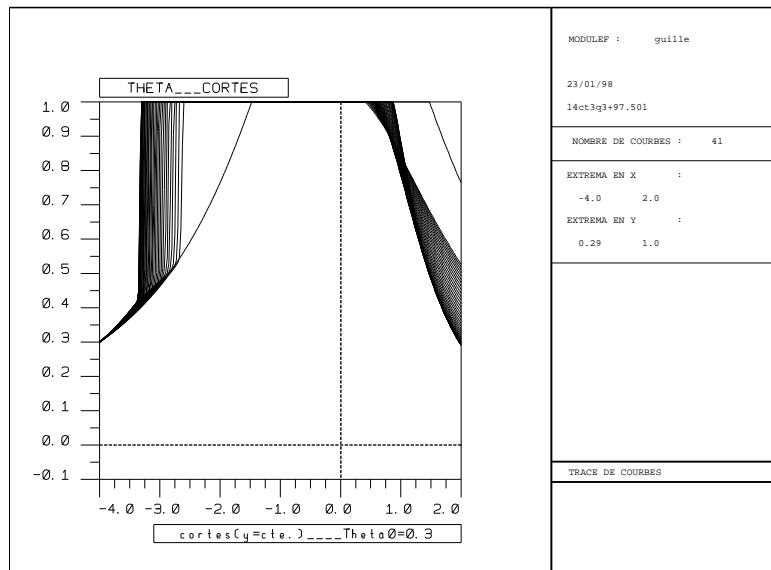


Figure 13.5 Saturation profile for  $\omega = 3$  and  $\theta_0 = 0.3$ .

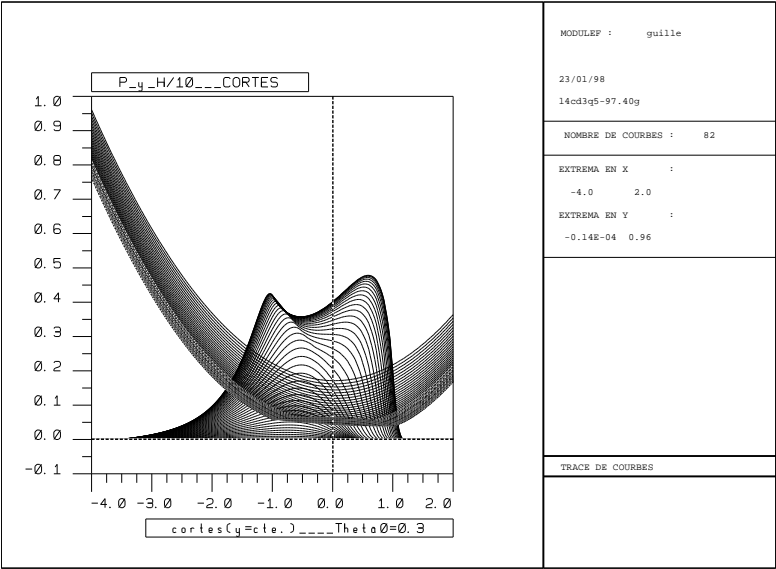


Figure 13.6 Pressure and gap profiles for  $\omega = 5$  and  $\theta_0 = 0.3$ .

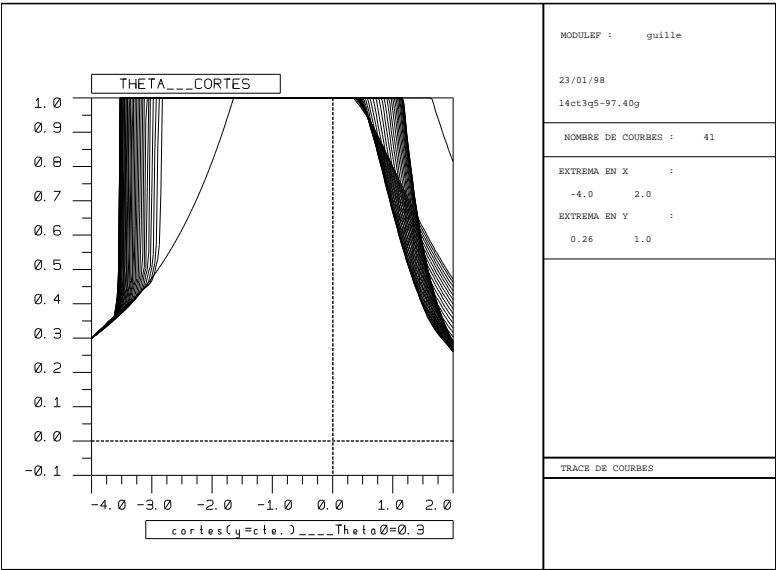


Figure 13.7 Saturation profile for  $\omega = 5$  and  $\theta_0 = 0.3$ .



the execution time significantly, which will allow us to obtain new numerical results for some tests whose CPU cost is extremely high to be run on a standard workstation.

As future work, we want to check the scalability of our parallel algorithm by running it on a higher number of processors. To do that, we intend to adapt the parallel code to the architecture of the Cray T3E multiprocessor. This way, numerical results for even finer finite element meshes could be computed.

## Acknowledgments

We gratefully thank CESGA (Centro de Supercomputación de Galicia, Santiago de Compostela, Spain) for providing access to the Fujitsu AP3000. This work was partially supported by Research Projects of Galician Government (XUGA 32201B97), Spanish Government (CICYT TIC96-1125-C03), D.G.E.S. (PB96-0341-C02) and European Union (1FD97-0118-C02).

## References

- [1] Barret R., Berry M., Chan T., Demmel J.W., Donato J., Dongarra J., Eijkhout V., Pozo R., Romine C., van der Vorst H.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Pub. (1994)
- [2] Cameron A.: *Basic Lubrication Theory*, John Wiley and Sons, Chichester (1981)
- [3] Durany J., García G., Vázquez C.: *Numerical Computation of Free Boundary Problems in Elastohydrodynamic Lubrication*, Appl. Math. Modelling, 20 (1996) 104-113
- [4] Durany J., García G., Vázquez C.: *Numerical Solution of a Reynolds-Hertz Coupled Problem with Nonlocal Constraint*, Scientific Computation Modelling and Applied Mathematics (Eds Sydow) Wissenschaft and Technik Verlag (1997) 615-620
- [5] Duff I.S., Erisman A.M., Reid J.K.: *Direct Methods for Sparse Matrices*, Clarendon Press (1986)
- [6] Gupta A., Gustavson F., Joshi M., Karypis G., Kumar V.: *Design and Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems*, Proceedings of 5th SIAM Conference on Parallel Processing, SIAM (March 1997)
- [7] Arenaz M., Doallo R., García G., Vázquez C.: *High Performance Computing of a New Numerical Algorithm for an Industrial Problem in Tribology*, Proceedings of 3rd Int'l Meeting on Vector and Parallel Processing, VECPAR'98, Porto, Portugal (June 1998) 1021-1034. Also selected to appear in Lecture Notes in Computer Science
- [8] Wolfe M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1996)



## 14 PARALLEL COMPUTING OF CAVITY FLOW BY FINITE ELEMENT METHOD<sup>\*</sup>

Hui Wan, Shanwu Wang, Yanxing Wang,  
Xiyun Lu, Lixian Zhuang

*Department of Modern Mechanics,  
University of Science and Technology of China  
Hefei, Anhui, 230026, P. R. China*

**Abstract:** A numerical algorithm applying equal order linear finite element and fractional four step methods is used for the analyses of incompressible fluid. The parallel computing is involved and the computation speed and precision are largely improved.

### 14.1 INTRODUCTION

It's reported in the literature that Finite Element Method (FEM) can be successfully applied to solving the Navier-Stokes equations, which is the key problem of Computational Fluid Dynamics (CFD). The major advantages of FEM are its ease in handling arbitrary or complex geometries, the ability to naturally incorporate differential-type boundary conditions and that any unstructured meshes can be treated without additional efforts. In particular, the solution of incompressible fluid flow can be performed efficiently in many practical cases. Many approaches have been proposed in the last decade to solve the incompressible Navier-Stokes equations. Among these approaches, stream function–vorticity formulation and velocity-pressure (primitive variable) formulation are often used. The former can not be extended to 3-d problem, so its use is largely limited. The methods by using the primitive variables are considered to be most important and are widely studied because the variables have obvious physical meaning and the form of equation provides a direct extension to three dimensions.

However, the primitive variable formulations have difficulties in the calculation of pressure, which is implicitly coupled with the divergence-free constraint on the velocity. Comparing with calculation of compressible flow, the continuity constraint

---

<sup>\*</sup> This work is supported by the National Natural Science Fund No. 19982009, by Fluid Mechanics Parallel Station of Department of Modern Mechanics of University of Science and Technology of China.

of incompressible Navier-Stokes equations prohibits time integration in a straightforward manner. So the numerical simulation of incompressible flow is a mathematically complex problem.

Moreover, for the large amount of matrix operation in the Finite Element Method, it needs far more computer resources than Finite Difference Method does. Consequently, requirements of larger computer memory and computing time by FEM overshadowed its advantages mentioned above.

Since the practical application of the first Array computer ILLIAC IV in 1975, Vector Computers and Parallel Computers have played a more and more important role in the scientific and engineering computations. In this work, MPI computation is performed in the simulation on the incompressible 3-d cavity flow, which is solved by a four step fractional method, in order to lower the computation time. The three-dimensional, time-dependent Navier-Stokes are numerically integrated by a time-split Galerkin Finite Element Method.

## 14.2 MATHEMATICAL FORMULATION AND NUMERICAL METHODS

### 14.2.1 Governing Equations

Unsteady three-dimensional viscous incompressible flow is described by the Navies-Stokes equations written in the tensor notation and dimensionless form as

$$\frac{\partial u_i}{\partial x_i} = 0 \quad (14.1)$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{\text{Re}} \frac{\partial e_{ij}}{\partial x_j} \quad (14.2)$$

In the above equations,  $t$  is the time,  $x_i$  is the Cartesian coordinate,  $u_i$  is the corresponding velocity component,  $p$  is the pressure, the viscous stress tensor  $e_{ij} = u_{i,j} + u_{j,i}$ , and  $\text{Re}$  is the Reynolds number.

### 14.2.2 Boundary Condition and Initial Condition

The boundary conditions are:

$$u_i|_{\Gamma_1} = b_i, \quad \tau_{ij} n_j|_{\Gamma_2} = t_i \quad (14.3)$$

in which,  $\Gamma_1$  and  $\Gamma_2$  are two subsets of the piecewise smooth domain boundary  $\Gamma$  with  $\Gamma_1 \cup \Gamma_2 = \Gamma$  and  $\Gamma_1 \cap \Gamma_2 = \emptyset$ ;  $b_i$  is the velocity component on the Dirichlet

boundary  $\Gamma_1$ , and  $t_i$  is the traction vector on the Neumann boundary  $\Gamma_2$ , both of which are given data;  $n_i$  is the outward unit vector normal to the boundary.

In the computation of a 3-d cavity problem, the top surface is a plane driven by lid so that  $u|_{y=1} = 1, v|_{y=1} = w|_{y=1} = 0$ , on the bottom and side walls, we have  $u = v = w = 0$ .

The initial conditions is  $\vec{u}(\vec{x}, 0) = \vec{u}_0(\vec{x})$ , which satisfies:

$$\nabla \cdot \vec{u}_0 = 0. \quad (14.4)$$

In computation of this paper,  $\vec{u}_0$  is set to be 0, that is, the driving lid moves suddenly at initial.

### 14.2.3 Fractional Method

Unlike most fractional methods mentioned in the previous literature, in which the pressure gradient term is de-coupled from the Navier-Stokes equation, In this paper, the pressure is still coupled with convection and diffusion. In this case, the velocity at the intermediate steps does not necessarily satisfy the continuity equation. Then, the new pressure is obtained by incorporating the continuity equation and the velocity at the new step is finally corrected by the pressure gradient to meet the requirement of mass conservation. The explicit fractional method is used for the convenience of transferring the information between sub-domains, which is decomposed from overall zone for parallel computation.

The fractional steps are written as follows:

$$\frac{u_i^{(1)} - u_i^n}{\Delta t} + u_j^{(n)} u_{i,j}^{(n)} = -p_{,i}^n + \frac{1}{\text{Re}} \tau_{ij,j}^n \quad (14.5)$$

$$\frac{u_i^{(2)} - u_i^{(1)}}{\Delta t} = p_{,i}^n \quad (14.6)$$

$$p_{,ii}^{n+1} = \frac{u_{i,i}^{(2)}}{\Delta t} \quad (14.7)$$

$$\frac{u_i^{n+1} - u_i^{(2)}}{\Delta t} = -p_{,i}^{n+1} \quad (14.8)$$

### 14.2.4 Finite Element Formulation of Pressure Equation

Pressure Poisson Equation will be solved in conventional fractional methods in which an artificial pressure boundary condition is needed. Thus the difficulty of the computation is raised. H.G.Choi, H.Choi and J.Y.Yoo [1] proposed a method to overcome this disadvantage in which, the pressure is determined by another pressure

equation, which can be derived from the Pressure Poisson equation and the incompressible constraint.

The Galerkin Formulation of continuity equation is:

$$\int_{\Omega} \phi_j u_{i,i}^{n+1} d\Omega = 0 \quad (14.9)$$

On account of  $(u_i \phi_j)_{,i} = \phi_{j,i} u_i + \phi_j u_{i,i}$  and the condition of divergence free, we can rewrite (14.9) as

$$\int_{\Omega} \phi_{j,i} u_i^{n+1} d\Omega = \int_{\Gamma} \phi_j u_i^{n+1} n_i d\Gamma \quad (14.10)$$

From (14.6) and (14.8) we get,

$$u_i^{n+1} = u_i^{(1)} - o(\Delta t^2) \quad (14.11)$$

Substituting Eq. (14.8) and (14.11) into (14.10) we obtain the following pressure equation, which takes the place of pressure Poisson equation

$$\Delta t \int_{\Omega} \phi_{j,i} p_{,i}^{n+1} d\Omega = \int_{\Omega} \phi_{j,i} u_i^{(2)} d\Omega - \int_{\Gamma} \phi_j u_i^{(1)} n_i d\Gamma \quad (14.12c)$$

## 14.2.5 Finite Element Formulation of Motion Equations by Fractional Methods

Here Galerkin method is used. The weak formulation of the momentum equation resulted from multiplying a weight function and integrating over the spatial domain can be written as follows:

$$\int_{\Omega} \left[ \phi_j (\dot{u}_j + u_k u_{j,k} + p_{,j}) + \frac{1}{\text{Re}} \tau_{ij} \phi_{i,j} \right] d\Omega - \int_{\Gamma_2} t_i \phi_i d\Gamma_2 = 0$$

where  $\phi_j$  is the weighting function, which has the following shape:

$$\phi_j = \frac{1}{8} (1 + \xi_j \xi) (1 + \eta_j \eta) (1 + \zeta_j \zeta), \text{ where } j = 1, 8 \text{ for a hexahedron element.}$$

The Galerkin formulation of four step fractional methods can be written as:

$$\int_{\Omega} \left[ \phi_k \left( \frac{u_i^{(1)} - u_i^n}{\Delta t} + u_j^n u_{i,j}^{(n)} + p_{,i}^n \right) + \frac{1}{\text{Re}} \tau_{ij}^{(n)} \phi_{k,j} \right] d\Omega - \frac{1}{\text{Re}} \int_{\Gamma_2} t_i^{(n)} \phi_k d\Gamma_2 = 0 \quad (14.12a)$$

$$\int_{\Omega} \phi_k \frac{u_i^{(2)} - u_i^{(1)}}{\Delta t} d\Omega = \int_{\Omega} \phi_k p_{,i}^n d\Omega \quad (14.12b)$$

$$\Delta t \int_{\Omega} \phi_{j,i} p_{,i}^{n+1} d\Omega = \int_{\Omega} \phi_{j,i} u_i^{(2)} d\Omega - \int_{\Gamma} \phi_j u_i^{(1)} n_i d\Gamma \quad (14.12c)$$

$$\int_{\Omega} \phi_k \frac{u_i^{(n+1)} - u_i^{(2)}}{\Delta t} d\Omega = - \int_{\Omega} \phi_k p_{,i}^{n+1} d\Omega \quad (14.12d)$$

By solving Eq. (14.12a), (14.12b), (14.12c), the intermediate velocity and the pressure are obtained. Then, using Eq. (14.12d), one can find the velocity of the next time step.

### **14.3 PARALLEL IMPLEMENTATION**

The parallel computation is implemented on Fluid Dynamics MPI Parallel Station (FDMPS) in the Department of Modern Mechanics of USTC. It is a small-scale parallel station, which contains following technical features.

#### **14.3.1 Hardware of FDMPS**

The FDMPS is a distributed memory, loosely coupled message passing parallel processing system. It is made up of 6 nodes (PII PCs). All the nodes are connected by SuperStack II Switch 3000 TX 8 port. The topology of this LAN is star type. The LAN obeys TCP/IP protocol, and its data transferring speed is 100 Mb/s. Each computing node is a PII PC, the frequency of the chip is 266M HZ. It has 128M SDRAM

The system service nodes are provided for managing the system resources. Two system service nodes charge the system initialization and the hardware failure diagnosis and I/O.

#### **14.3.2 Software of FDMPS**

The operating system of FDMPS is based on LINUX (Version RedHat 5.1). The parallel programming tool is MPI (Message Passing interface). Both Fortran and C can be used in this environment.

#### **14.3.3 Parallel Computation of 3-D Cavity**

The program is written in Fortran77 with Message Passing Interface (MPI) and implemented on 4 Processors of the Fluid Dynamics Parallel Station of USTC. The flow field domain is divided into 4 sub-domains by Y and Z directions. In computation, SIMD model is used, that is, each node calculates in one domain. The data of the domain boundary are exchanged by Message Passing. Overlapped cell in each sub-domain is needed to store inner-boundary information of sub-domains (Fig.14.1, 14.2). The overlapped zone has two layers meshes in this work. The number of overlapped layers is decided by the dissociation formulation of equations.

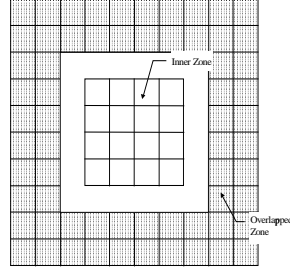


Figure 14.1 One Sub-domain (X-Y direction)

$$\begin{aligned} Q_{N_{\max}}^*|_0 &\leftarrow Q_3^*|_1 \quad \& Q_{N_{\max}-1}^*|_0 \leftarrow Q_2^*|_1 \\ Q_0^*|_1 &\leftarrow Q_{N_{\max}-3}^*|_0 \quad \& Q_1^*|_1 \leftarrow Q_{N_{\max}-1}^*|_0 \end{aligned}$$

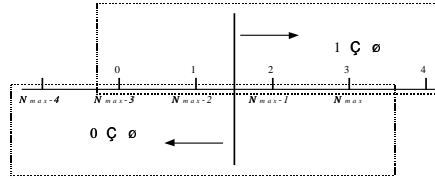


Figure 14.2 Sketch map of data communication and mesh overlap

## 14.4 RESULTS AND DISCUSSIONS ON 3D CAVITY FLOW

Here we calculated the lid-driven cavity flow, which is a fundamental problem of Fluid Dynamics and is often used as a benchmark. The Reynolds numbers are 1000 and 5000 respectively. The results are in correspondence with that of computed by C. Nonino and G. Comini, [2]. The computational grid employed in this simulation consists of  $28 \times 28 \times 28$  linear elements, with smaller size near the boundary walls. Time step 0.01 is used.

Graphical results concerning the three mutually orthogonal mid-planes are shown in Figure 14.3. We showed three pseudo-streamlines, which are plotted by the velocity components of the mid-plane. They have no much physical meaning, but they may be helpful in analysis of the flow structure. Figure 14.4 shows the velocity vectors of mid-planes. A main vortex is generated on the symmetry plane. For the existence of the transversal vortices in the other two directions, the intensity of this main vortex is



#### *PARALLEL COMPUTING OF CAVITY FLOW*

lower than that of the vortex of 2d-cavity lid driven problem in the same Reynolds numbers.

The pressure contours of the three mid-planes are showed in Figure 14.5. It's reported that the oscillation occurs in the pressure contour when Reynolds number high to 1000 and when the calculation is executed by Galerkin finite element method. This is because the Galerkin FEM bears a resemblance to the central difference in the property of negative dissipation. In this paper, the oscillation of pressure doesn't occur when Reynolds number is 1000. This maybe because in the calculation of the inner flow, the peripheral boundaries can limit the numerical dissipation to some extent. In the calculation of Reynolds number 5000, pressure oscillation occurs, as showed in Figure 14.8. In Figure 14.6, we can see the pseudo-streamlines of the different planes that are parallel to the symmetry plane. This Figure shows largely different flow structures lengthways. Figure 14.7 shows the three dimensional streamline of the cavity.

Figure 14.3 Pseudo-Streamline,  $Re=1000$ ,  $t=15s$ , (1) x-y mid-plane; (2) x-z mid-plane; (3) z-y mid-plane

Figure 14.4 Velocity vectors on three mid-planes

Figure 14.5 Pressure contours on three mid-planes.

*PARALLEL COMPUTING OF CAVITY FLOW*

Figure 14.6 Pseudo-streamline on different x-y plane,  $t=15$ ,  $Re=1000$

Figure 14.7 Three-dimensional streamlines on  $t=15s$ .



Figure 14.8  $Re=5000$ ,  $t=15s$ , velocity vectors and pressure contours.  
(a) x-y mid-plane; (b) x-z mid-plane; (c) z-y mid-plane;  
(d) x-y mid-plane pressure contour.

## 14.5 CONCLUDING REMARKS ON MPI

Solving the Navier-Stokes Equations for the incompressible flow, by parallel computational methods, we simulated the three-dimensional cavity lid-driven problem successfully and the speedup is almost linear.

It has been confirmed that MPI parallel tools can be readily realized on PC system. It doesn't need too much investment on hardware to gain high performances of computation. It's not only suitable for the massively parallel environment, but also can be used in some small-scale system. Such kind of system can meet the needs of different users. According to the international standard, MPI is a portable and scalable tool could be easily used in different cases such as different operating system and different machines.

The debugging of parallel program is relatively more difficult than serial program. Though there are already a lot of debug tools, it seems that they are not very robust. So developing more practical debugging tools is an task for both users and developers.

When breaking up one task into small tasks by Data Decomposition (Domain Decomposition) method, one should think the directions of the decomposition over. For example, for 4 processors, the domain can be delimited to be  $1 \times 4$ , or  $2 \times 2$ , the efficiency may be different. How to decompose a domain effectively depends on the parallel architecture, the parallel program and the task itself.

In the computation of this paper, for the convenience of transferring data of overlapped zone, the explicit formulation of the equations is applied, which needs small time step for advancing. So the advantage brought by parallel computation is impaired to some extent. To solve this problem, the new functional parallel formulation should be investigated further. This Decomposition method would take the characteristic of Computational Fluid Dynamics into account, especially should make full use the large amount of implicit stable formulation of CFD.

## Reference

- [1] H.G.Choi, H.Choi, J.Y.Yoo, "A Fractional Four-step Finite Element Formulation of the Unsteady Incompressible Navier-Stokes Equations using SUPG and Linear Equal-order Element Methods", *Comput. Methods Appl. Mech. Engrg.* 143 (1997) 333-348.
- [2] C.Nonino, G.Comini "An Equal-Order Velocity -Pressure Algorithm for Incompressible Thermal Flows", *Numerical heat Transfer, Part B*, 32: 1-15, 1997
- [3] Carriero, Nicholas and Gelernter, David, "How to Write Parallel Programs - A First Course". MIT Press, Cambridge, Massachusetts
- [4] Shanwu Wang, *Numerical Simulation on flow past 2-d and 3-d airfoil at high attack of angle*. Master degree thesis of Univ. of Sci. & Tech. of China, 1998/11/29
- [5] J.G.Rice and R.J.Schnipke, An equal order velocity pressure formulation that does not exhibit spurious pressure modes, *Comput. Methods Appl. Mech. Engrg.* 58 (1986) 135-149

## ELECTROMAGNETIC SCATTERING WITH THE BOUNDARY INTEGRAL METHOD ON MIMD SYSTEMS

Thierry Jacques, Laurent Nicolas, Christian Vollaire

*CEGELY - UPRESA CNRS 5005  
Ecole Centrale de Lyon- B.P. 163  
69130 Ecully cedex- France*

{Thierry.Jacques, Laurent.Nicolas, Christian.Vollaire}@eea.ec-lyon.fr

**Abstract:** This chapter deals with parallel computation in electromagnetics. It is first demonstrated that electrical engineering is far behind other engineering disciplines in terms of use of parallel computers. Parallel numerical methods for field calculation are then briefly reviewed. The parallel implementation of a boundary integral formulation is then presented. It is used to model electromagnetic scattering by perfect electric conducting or perfect dielectric bodies. Because this method requires large memory storage, it is implemented on a distributed memory parallel machine. The assembling is performed by nodal contribution, and the BiCGStab solver is used for solving. Parallel performances are finally analyzed with several large problems.

## **15.1 INTRODUCTION**

Numerical computation is more and more used by scientists to modelize physical phenomena or to optimize existing systems. Nowadays, only parallel computers are able to provide the required power to solve today's realistic problems. There are two reasons : large memory is required because of a large amount of data, or speed is required to obtain the solution.

In this paper, we are interested in the frequency domain boundary element method (BEM) in order to compute electromagnetic scattering problems: the objective is to show how the BEM has been implemented efficiently on a distributed memory parallel computer.

The appearance of parallel computers has caused a revolution in all scientific domains. However, electrical engineering seems to be far behind other computational disciplines : this is shown in the first section. A brief review of several parallel implementations of the numerical methods used in computational electromagnetics is then given. It is also shown that only a few attempts to parallelize the boundary element method have been made. This method shows however several advantages: only the material interfaces have to be discretized, and the decrease of the field at infinity is implicitly taken into account. On the other hand, it generates a full non-Hermitian matrix which requires large memory capabilities. In the next sections, the formulation that we use and the way that we have implemented it are described.

## **15.2 IMPACT OF THE PARALLEL COMPUTING IN THE ELECTRICAL ENGINEERING**

In recent years, several works in parallel computational electromagnetics have been reported, especially since new distributed memory architectures appeared. It seems however that electrical engineering is far behind other domains in parallel computation, in the same manner than computational electromagnetics was behind computational mechanics in early 70's. For example, 3D optimization using genetic algorithms for Navier-Stokes computations is today currently performed in fluid dynamics [1]. The reason for this delay is certainly due to the fact that field problems are generally

neither confined to a structure nor set in very regular domains. The geometries involve substantial detail and the domains of study have often to be considered to infinity.

As proof, table 15.1 shows the evolution since 1993 of the number of papers dedicated to parallel computation in two reference conferences dealing with the computation of electromagnetic fields: COMPUMAG (Conference on the Computation of Electromagnetic Fields) and CEFC (IEEE Conference on Electromagnetic Field Computation). In the same table the number of papers on parallel computation is given for ECCOMAS'96, which is the European Conference on Numerical Methods in Engineering (Paris, 9/9/96 – 9/13/96). In this last one, papers are divided into two classes: Fluid Mechanics, and other Engineering sciences. Obviously a great effort has to be done in our domain in order to hoist parallel computation to its real place. Consequently, one can predict a wide expansion of the use of parallel computation in electrical engineering.

Table 15.1 Comparison of number of papers on parallel computation in different conferences on numerical methods (Eng. Sciences : Engineering Sciences).

Conference	COMPUMAG'93	CEFC'94	COMPUMAG'95	CEFC'96
number of papers	302	344	366	420
papers on parallel computation	4	4	3	6
ratio	1,3%	1,2%	0,8%	1,4%

Conference	COMPUMAG'97	CEFC'98	ECCOMAS'96 Fluid Mechanics	ECCOMAS'96 Eng. Sciences
number of papers	419	394	168	160
papers on parallel computation	9	4	25	12
ratio	2,1%	1,0%	15,0%	8,0%

Numerical methods which are the most widely used in electromagnetic field computation are the Finite Difference Time Domain (FDTD) and the Finite Volume Time Domain (FVTD) methods, which have been developed especially for aeronautics. Yee first proposed to solve the Maxwell's equations by using the finite-difference method with a discretization in time and in space [2]. The FDTD is well adapted to electromagnetic wave



phenomena, including scattering, penetration, coupling and interaction studies. The Maxwell's equations are solved in a volume region of space that fully contains the structure being modeled. The studied domain is bounded by a surface that absorbs the waves spreading at infinity. Several types of condition may be applied on this boundary surface : absorbing boundary condition (ABC) or perfect matching layers (PML). Several examples of implementation of FDTD have been reported in the literature [3,4]. All show that the FDTD is easily implementable on distributed memory parallel computers. To obtain good parallel efficiency, it is however necessary to adapt the number of processors to the size of the computed problem: in other words, there is an optimum to be found in the number of FDTD cells per processor.

The potential of the FVTD method comes from the fact that it is based on proven computational fluid dynamics techniques early developed. It allows to study scattering, radiation, electromagnetic compatibility, shielding and interference problems. The FVTD method casts the Maxwell's curl equations in a conservation form. In [5, 6] an explicit time discretization that is based on the Lax-Wendroff upwind scheme is developed. A domain decomposition technique is used: each processor has to compute the solution on a specific sub-region. Complex problems, such as the scattering of a full scale fighter geometry at 500 MHz frequency, may be computed, showing the efficiency of such an approach.

The finite element method (FEM) is attractive to model 2D and 3D problems because of its adaptability for designing complex problems. But the modeling of realistic devices requires the solution of large problems which can be afforded only by parallel computation. Several examples of parallelization of the finite element method for electromagnetics have been recently reported [7, 8].

### **15.3 INTEGRAL EQUATION METHOD IN THE PARALLEL COMPUTATION**

There is no significant distinction between Laplace and Helmholtz problems for the parallelization of frequency-domain integral equation methods: both involve the formation of a dense matrix, followed by the solution of the

matrix equation which is cost dominant. The two first parts of this section are dedicated to such frequency-domain problems: the method of moments for high frequency, and the boundary equation method for electrostatics. In a third part, the parallel implementation of a time-domain integral equation is described.

### **15.3.1 Parallel implementation of the method of moments**

The method of moments (Mom) has been proposed by Harrington in the mid 1960's [9]. It may be applied to electrostatics or high frequency problems and mainly to scattering and penetration problems. The prediction of the scattering of 3D arbitrarily shaped electrically large targets is an important problem, especially for aeronautics. For such applications, the parallel computation is essential.

In [10], the dense matrix is distributed over the processors and is factored into lower and upper triangular matrices. The efficiency of the LU factorization decreases with the size of the problem because of the communications between the processors. It is shown that the total computation time reaches a saturation when the number of processors exceeds a limit related the problem size. It tends to prove that there is an optimum to find between the number of processors and the problem size.

Large problems, such as those computed with the FVTD method, can be modeled, showing that the Mom is also powerful.

### **15.3.2 Example of the boundary element method for static problems**

In [11], a method to compute in parallel the coupling capacitances of multi-conductor 3D structures on a network of SPARC workstations connected by an Ethernet network is presented. The formulation results in a set of two integral equations which is solved for the unknown surface charge density. Applying the boundary element method (BEM) for discretizing both of these integral equations with constant basis functions leads to a linear equation system which is dense. The implementation on the cluster of workstations is made using PVM, and the algorithm works using the master-slave principle. The geometry and material information is transported to each workstation. Because each row and column respectively is completely independent of all others, the matrix filling is performed in a column block oriented way. Hence it is done without any communication. The solving of the non-

symmetric dense matrix is performed with a GMRES algorithm with Jacobi preconditioning. This algorithm has been chosen because it has a minimal communication for vector updates (only one per iteration) and a good convergence rate. A load balancing algorithm is used in order to get a better distribution of load along the workstations: the master holds information on the physical memory of all involved workstations and only schedules new tasks to a requesting slave if it is not in a heavy load state. The inhomogeneous structure of the workstation cluster can be better exploited, resulting in an improved speedup.

### 15.3.3 Time-domain integral equation

If parallel computation of frequency-domain integral equation methods has received some attention, only one attempt has been made to parallelize a time-domain integral equation approach [12]. In this paper, the magnetic field integral equation is addressed. The magnetic field at a point and at time  $t$  depends on all the points on the boundary of objects from the time 0 to the time  $t$ . This yields a matrix equation of the form :

$$[\mathbf{a}]\{H^k\} = [\mathbf{b}]\{H^{k-1} | H^{k-2} | \dots | H^0\}$$

where  $\{H^k\}$  is the list of historical values of the field at the nodes on the surface. Matrix  $\mathbf{b}$  is very sparse, since it is constituted of historical boundary field values which contribute in each case to the field at a particular field node. Once  $\mathbf{b}$  has been computed, the right hand side (RHS) is then calculated at each time step by a large matrix-vector multiplication. The final matrix equation is then solved at each time step. Preliminary tests on a scalar workstation show that the RHS formation is the dominant cost in computation time, and it tends to increase when the size of the problem increases. Consequently, the strategy for parallelizing the code has been guided by this observation, and the most parallelization effort has been devoted to this stage. The formation of the matrix  $\mathbf{a}$  has been implemented in parallel : each processor computes a part of the matrix. Concerning the RHS formation, a set of field nodes is allocated to each processor and the right hand side is evaluated partially by each processor from its set of boundary

nodes. The matrix solution is performed in parallel using the conjugate gradient algorithm. Good parallel efficiency is obtained for both the matrix formation and the RHS formation. On the other hand, the parallelization of the matrix solution is poor.

## **15.4 BOUNDARY INTEGRAL FORMULATION FOR ELECTROMAGNETIC SCATTERING PROBLEMS**

It has been shown that only a few attempts to parallelize the boundary element method have been made. The objective of this paper is to describe how we have implemented the BEM on a distributed memory parallel computer in order to model electromagnetic scattering problems.

Let's assume that an incident wave (frequency  $f$ , pulsation  $\omega$ ) is perturbed by the object of boundary  $\Sigma$ .

Fig.15.1 Description of the problem. The electromagnetic properties of the material are its conductivity ( $\sigma$ ), its permittivity ( $\epsilon$ ) and its permeability ( $\mu$ ).

### **15.4.1 Perfect electric conducting body ( $\sigma=\infty$ , $\epsilon_1=\epsilon_0$ )**

Since the skin depth is very small compared to the dimensions of the perfect electric conducting (PEC) body, the fields  $\mathbf{H}$  et  $\mathbf{E}$  are null inside the region  $\Gamma_1$ . By applying the Green's theorem to Maxwell's equations into the region  $\Gamma_0$ , the following equation is obtained [13]:

$$\mathbf{J}(\mathbf{x}) = 2\mathbf{n} \times \mathbf{H}^{\text{inc}}(\mathbf{x}) + \frac{\mathbf{n}}{2\pi} \times \oint_{\Sigma} \mathbf{J} \times \nabla \Phi_0 \, ds$$

where  $\Phi_0(\mathbf{x}, \mathbf{y}) = e^{-jk_0|\mathbf{x}-\mathbf{y}|} / |\mathbf{x}-\mathbf{y}|$  is the Green's function,  $\mathbf{J}$  is vector current density, and  $\mathbf{n}$  is the exterior normal vector. This formulation leads to

three complex degrees of freedom per node, corresponding to three rows in the matrix.

### 15.4.2 Perfect dielectric body ( $\sigma=0$ , $\epsilon_1 \neq \epsilon_0$ , $\mu_1 \neq \mu_0$ )

By applying Green's theorem in the inner and outer regions and by using the boundary conditions at infinity and at material interfaces, the following equations may be obtained [14]:

$$\begin{aligned} \frac{1+\mu}{2} \mathbf{J} &= \mathbf{n} \times \mathbf{H}^{\text{inc}} + \frac{\mathbf{n}}{4\pi} \times \iint_{\Sigma} k_0 \mathbf{K}(\Phi_0 - \mu \epsilon \Phi_1) + \mu H_n (\nabla \Phi_0 - \nabla \Phi_1) + \mathbf{J} \times (\nabla \Phi_0 - \mu \nabla \Phi_1) ds \\ \frac{1+\mu}{2} H_n &= \mathbf{n} \cdot \mathbf{H}^{\text{inc}} + \frac{\mathbf{n}}{4\pi} \iint_{\Sigma} k_0 \mathbf{K}(\Phi_0 - \epsilon \Phi_1) + H_n (\mu \nabla \Phi_0 - \nabla \Phi_1) + \mathbf{J} \times (\nabla \Phi_0 - \nabla \Phi_1) ds \\ \frac{1+\epsilon}{2} \mathbf{K} &= \alpha \mathbf{n} \times \mathbf{E}^{\text{inc}} + \frac{\mathbf{n}}{4\pi} \times \iint_{\Sigma} k_0 \mathbf{J}(\Phi_0 - \mu \epsilon \Phi_1) + \epsilon E_n (\nabla \Phi_0 - \nabla \Phi_1) + \mathbf{K} \times (\nabla \Phi_0 - \epsilon \nabla \Phi_1) ds \\ \frac{1+\epsilon}{2} E_n &= \alpha \mathbf{n} \cdot \mathbf{E}^{\text{inc}} + \frac{\mathbf{n}}{4\pi} \iint_{\Sigma} k_0 \mathbf{J}(\Phi_0 - \mu \Phi_1) + E_n (\epsilon \nabla \Phi_0 - \nabla \Phi_1) + \mathbf{K} \times (\nabla \Phi_0 - \nabla \Phi_1) ds \end{aligned}$$

where  $\mu = \mu_1/\mu_0$ ,  $\epsilon = \epsilon_1/\epsilon_0$ ,  $\alpha = j\sqrt{\epsilon_0/\mu_0}$ ,  $\mathbf{J} = \mathbf{n} \times \mathbf{H}$ ,  $H_n = \mathbf{H}_1 \cdot \mathbf{n}$ ,  $\mathbf{K} = \alpha \mathbf{n} \times \mathbf{E}_1$  and  $E_n = \alpha \mathbf{E}_1 \cdot \mathbf{n}$ .  $\Phi_0$  and  $\Phi_1$  are Green's function of respective wave number  $k_0$  and  $k_1$ . These variables and the coefficients of the matrix are of the same order, leading to a better conditioning of the matrix. This formulation leads to eight complex degrees of freedom per node.

### 15.4.3 Numerical implementation

The boundary of the objet is meshed with second order surface finite elements. Ten nodes per wavelength are required for a good accuracy ( $\lambda = 2\pi/k_0$  for a perfect electrical conducting body,  $\lambda = 2\pi/k_1$  for a dielectric body). If this condition is not respected, it may cause convergence problems during the solving stage. Surface integrals are computed using Gauss integration. The function to be integrated varies in  $1/r^2$  where  $r$  is the distance between the calculation node and the integration point. The singularity is then computed analytically[15].

Half-discontinuous finite elements allow to treat geometrical discontinuities [16]: the points on the edges are separated and moved towards the interior of the elements.

## 15.5 PARALLEL COMPUTATION

For a  $n$ -nodes meshing, the storage of the matrix should require  $18n^2$  real data for a PEC body et  $128n^2$  real data for a perfect dielectric body. Parallel computation is then essential to compute realistic devices.

### 15.5.1 Assembling

The global system matrix is computed by node: each processor captures data (mesh and physical data) and computes a part of the lines of the matrix relative to the nodes. For the node  $i$  on a perfect electric conductor, three rows of the global matrix are computed. The contribution of the node  $j$  on the node  $i$  may be put on the following form :

$$\begin{bmatrix} -a_{jy}n_y - a_{jz}n_z & a_{jy}n_x & a_{jz}n_x \\ a_{jx}n_y & -a_{jx}n_x - a_{jz}n_z & a_{jz}n_y \\ a_{jx}n_z & a_{jy}n_z & -a_{jx}n_x - a_{jy}n_y \end{bmatrix}$$

where the vector  $(n_x, n_y, n_z)$  is the normal vector at node  $i$ . The  $3 \times 3$  submatrix corresponding to the contribution of node  $i$  on itself is a scalar matrix and is located on the diagonal of the global matrix. For each node, only the coefficients  $(a_{jx}, a_{jy}, a_{jz})$  are stored, such as the diagonal coefficient and the normal vector. This allows to reduce the storage by three, leading to  $6n^2$  real data. By using the same trick for a dielectric body, the storage may be reduced by eight, leading to  $16n^2$  real data.

Since the matrix is row-distributed, no communication between the processors is required during the assembling stage, and the speedup relative to this stage is optimal.

### 15.5.2 Solver

An iterative method is used because allowing to reduce the storage. Consequently, the minimal number of processors which is required to

compute a problem and the communication numbers are smaller than for a direct method.

### 15.5.3 Preconditioning

A diagonal preconditioning is used. The right preconditioning requires the broadcast and the storage of the diagonal, the computing of a intermediate vector by each processor after each multiplication matrix-vector. This intermediate vector has also to be kept in memory. On the other hand, the left preconditioning can be carried out in the same time as the multiplication matrix-vector. It requires less memory. Hence the left preconditioning is more efficient, and is preferably used.

### 15.5.4 Parallel algorithm of BiCGStab

We use the algorithm BiCGStab of H. Van der Vorst [17] to solve the problem  $Kax = Kb$ , where  $K$  is the preconditioning. At each iteration, two multiplications matrix-vector are required. The solution vector is distributed on each processor. The algorithm has been parallelized by minimizing the communications between processors:

Table 15.2 Parallelized version of the algorithm BiCGStab -■: sequential part, ○: parallel part.

```

x = 0
q = p0 = r0 = Kb
ω = 1
ρ = <q, r0>
Broadcast of vector q and ρ
while(ej > ε)
  ○ vj = K A rj
  ○ Φ1 = <q, vj>
    Broadcast of vector vj and of the local inner product
  ■ α = ρ ω / Φ1
  ■ sj = rj - α vj
  ○ tj = K A sj
  ○ ρ = -<q, tj>

```

$$\circ \phi_1 = \langle s_j, t_j \rangle$$

$$\circ \phi_2 = \langle t_j, t_j \rangle$$

$$\circ \phi_3 = \langle s_j, s_j \rangle$$

Broadcast of vector  $t_j$  and of 4 local inner products

$$\blacksquare \omega = \phi_1 / \phi_2$$

$$\circ x = x + \alpha p_j + \omega r_j$$

$$\blacksquare r_{j+1} = s_j - \omega t_j$$

$$\blacksquare \beta = \rho / \omega$$

$$\blacksquare p_{j+1} = r_{j+1} + \beta (p_j - \omega v_j)$$

$$\blacksquare e_j = \phi_3 - \omega^2 \phi_2$$

Before and after each broadcast performed using PVM, a synchronization between processors is carried out. For a perfect dielectric body, when the frequency increases, it is necessary to increase the mesh discretization. Because the algorithm BiCGStab shows some difficulty to converge, especially when using half-discontinuous elements, the algorithm BiCGStab(l)[18] is preferably used. When  $l$  increases, the algorithm is more robust. On the other hand, it requires more memory and it is less parallelizable than the BiCGStab.  $2 \times l$  message passing are then required at each iteration, corresponding to the  $2 \times l$  matrix-vector multiplications.

## 15.6 NUMERICAL RESULTS

The first test problem is the scattering of a 3 GHz plane wave by a PEC cylinder (fig. 15.2). The length of the cylinder goes from  $\lambda$  to  $6\lambda$ , and its radius is equal to  $0.5\lambda$  (where  $\lambda$  is the wavelength).

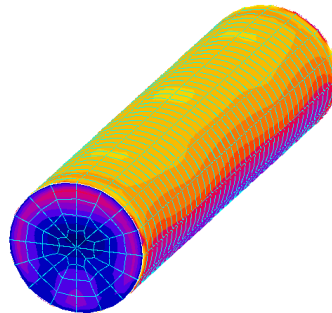


fig. 15.2. Scattering by a perfect electric conducting cylinder ( $L = 4\lambda$ ) - visualization of the surface current density  $J$  - 4662 degrees of freedom.



The second example is the scattering of a 900 MHz plane wave by a dielectric cylinder ( $\mu = 1$  and  $\epsilon = 2$ ) of same dimension (fig. 15.3).

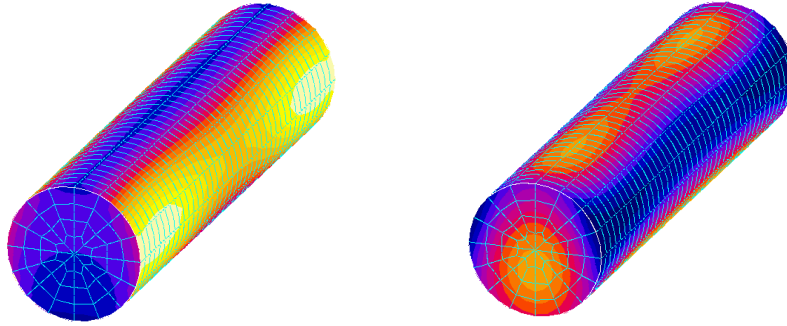


Fig. 15.3. Scattering by a dielectric cylinder ( $\mu=1$ ,  $\epsilon=2$ ) – Left: visualization of the electric field  $\mathbf{E}$  – Right: visualization of the magnetic field  $\mathbf{H}$  - 12432 degrees of freedom.

Both examples are computed on the Cray T3E of the IDRIS (Institut du Développement et des Ressources en Informatiques Scientifiques, CNRS). Results are presented in terms of parallel efficiency, since some problems cannot be computed on one processor: if the minimal number of processors which can be used is  $p$ , the reference computation time obtained on these  $p$  processors leads to a parallel efficiency equal to  $p$ .

Several computation times are presented in the table 15.2. As shown in fig. 15.4 and fig 15.5, the parallel efficiency remains high when the number of processors increases. Two reasons may be underlined. First the assembling stage remains dominating (from 70% to 85% of total time), with a speedup nearly optimal since no passage of message is required and the computing load is well distributed over the processors. Second, the main stage of the solver is the matrix-vector multiplication, which is parallelized. During this stage, the computing load is also well-balanced because each processor performs the same number of operations.

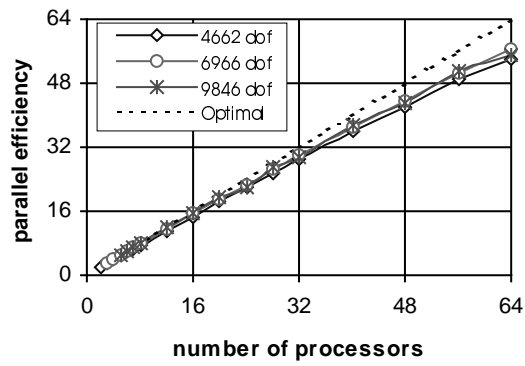


Fig. 15.4 Parallel efficiency for several number of degrees of freedom - Scattering of a 3 GHz plane wave by a PEC cylinder.

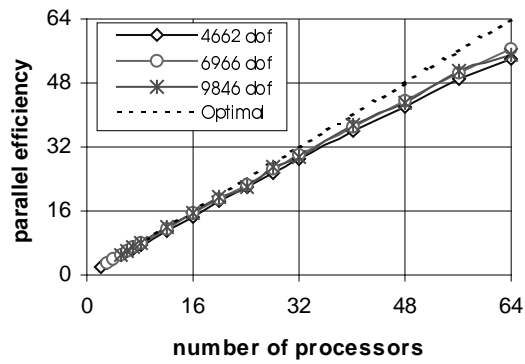


Fig. 15.4 Parallel efficiency for several number of degrees of freedom - Scattering of a 3 GHz plane wave by a PEC cylinder.

Table 15.2. Computation times (in s.) /number of iterations of BiCGStab(4) for the PEC cylinder and of BiCGStab(2) for the dielectric cylinder.

<b>PEC cylinder</b>			
Degrees of freedom	4662	6966	9846
12 processors	553 s. / 11 it.	1224 s. / 28 it.	2735 s. / 26 it.
20 processors	558 s. / 11 it.	1263 s. / 29 it.	2795 s. / 26 it.
32 processors	569 s. / 11 it.	1266 s. / 28 it.	2912 s. / 28 it.
48 processors	585 s. / 11 it.	1317 s. / 30 it.	3006 s. / 29 it.
64 processors	607 s. / 11 it.	1349 s. / 30 it.	3122 s. / 30 it.
<b>Dielectric cylinder</b>			
Degrees of freedom	12432	18576	26256
12 processors	1083 s. / 4 it.	2344 s. / 4 it.	4784 s. / 5 it.
20 processors	1086 s. / 4 it.	2396 s. / 4 it.	4852 s. / 5 it.
32 processors	1095 s. / 4 it.	2397 s. / 4 it.	4862 s. / 5 it.
48 processors	1111 s. / 4 it.	2423 s. / 4 it.	4921 s. / 5 it.
64 processors	1127 s. / 4 it.	2451 s. / 4 it.	4956 s. / 5 it.

## 15.7 CONCLUSION

A parallel version of the Boundary Element Method to solve electromagnetic scattering is presented. The mesh is distributed equally over the processors and the assembling of the full matrix is carried out by node. This allows to avoid message passing between processors during this stage. A parallel version of the BiCGStab(1) algorithm is used for the matrix solving. It is shown that this iterative solver is efficient. This is due to the fact that the matrix-vector multiplication is parallelized and constitutes its main stage.

## References

- [1] S. Obayashi, A. Oyama. Three-dimensional Aerodynamic Optimization with Genetic Algorithm. Computational Fluid Dynamics'96, John Wiley & Sons Ltd., pp 420-424, 1996
- [2] K.S. Yee. Numerical Solution of Initial Boundary-Value Problems involving Maxwell's Equations in Isotropic Media. IEEE Transactions on Antennas and Propagation, vol.AP-14, pp. 302-307, 1966.

- [3] V. Varadarajan, R. Mittra. Finite Difference Time Domain Analysis Using Distributing Computing. IEEE Microwave and Guided Wave letters. Vol. 4, pp. 144-145, May 1994.
- [4] Y. Lu, C.Y. Shen. A Domain Decomposition A Domain Decomposition Finite-Difference Method for Parallel Numerical Implementation of Time-Dependent Maxwell's Equations. IEEE Transaction. On Antennas and Propagation, vol. 45, no. 3, pp. 556-562, March 1997
- [5] A.C. Cangellaris. Time-Domain Finite Methods for Electromagnetic Wave Propagation and Scattering. IEEE Transaction on . Magnetic., vol. 27, no. 5, pp. 3780-3785, September 1991.
- [6] C. Rowell, V. Shankar, W.F. Hall, A. Mohammadian. Advances in Time-Domain CEM using Massively Parallel Architectures. 11th Annual Review of Progress in Applied Computational Electromagnetics, Monterey CA, March 20-25, 1995, pp. 839-846
- [7] M.L. Barton. Three-dimensional Magnetic Field computation on a Distributed Memory Parallel Processor. IEEE Transaction on Magnetic, vol.26, no. 2, pp. 834-836, March 1990.
- [8] C. Vollaie, L. Nicolas, A. Nicolas. Finite element and Absorbing Boundary Conditions for Scattering Problems on a parallel Distributed Memory Computer. IEEE Transaction on Magnetic, vol. 33, no. 2, pp. 1448-1451, March 1997.
- [9] R.F. Harrington. Field Computation by Moment Methods. MacMillan, New-York, 1968.
- [10] T. Cwik, J. Partee, J. Patterson. Method of Moment Solutions to Scattering Problems in a parallel Processing Environnement. IEEE Transaction on Magnetic, vol. 27, no. 5, pp. 3837-3840, September 1991.
- [11] C.F. Bryant, M.H. Roberts, C.W. Trowbridge. Implementing a Boundary Integral Method on a Transputer System. IEEE Transaction on Magnetic, vol. 26, no. 2, pp. 819-822, March 1990.
- [12] S.P. Walker, C.Y. Leung. Parallel Computation of Time-Domain Integral Equation Analyses of Electromagnetic Scattering and RCS. IEEE Transactions on Antennas and Propagation. Vol. 45.no.4. pp. 614-619, April 1997.
- [13] M.J. Bluck, S.P. Walker. Time-Domain BIE Analysis of Large Three-Dimensional Electromagnetic Scattering Problems. IEEE Transation on Antennas and Propagation, vol. 45, no. 5, pp. 894-901, May 1997.
- [14] E. Schlemmer., W.M. Rucker, W.M., K.R. Richter. Calcul des champs électromagnétiques diffractés par un obstacle diélectrique tridimensionnel par des éléments de frontière en régime transitoire. Journal Physique III, pp. 2115-2126, November 1992.

- [15] C.J. Huber, W. Rieger, M. Haas, W.M. Rucker. The numerical treatment of singular integrals in boundary element calculations, 7<sup>th</sup> International IGTE Symposium, 1996. pp. 368-373.
- [16] S. Rêgo, J.J. Do. Acoustic and Elastic Wave Scattering using Boundary Elements. Computational Mechanics Publications, UK, 1994.
- [17] H.A. Van der Vorst. BiCGStab: A fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. SIAM J.Science Statist. Comput., 13, pp. 631-644. 1992. Available on <http://www.math.ruu.nl/people/vorst/cgnotes/cgnotes.html>
- [18] G.L.G. Sleijpen, D.R. Fokkema. BiCGStab(l) for linear equations involving unsymmetric matrices with complex spectrum. ETNA, volume1, September 1993, pp. 11-32. Available on <http://www.math.ruu.nl/people/sleijpen>

# Index

- absorbing boundary condition, 179, 217
- additive, 55
- additive algorithm, 54
- additive schwarz algorithm, 178
- algorithmic blocking, 20
  
- backward substitution, 13
- balanced binary tree, 91
- BiCGStab, 223
- biconjugate gradient method, 64
- biorthogonalization algorithm, 64
- BLAS routines, 5
- block cyclic data distribution, 20
- block recursive ordering, 78
- boundary element method, 215
- boundary integral method, 215
- bounded polyhedral domain, 106
- broadcast, 35, 224
- broadcast time, 96
- butcher array, 37
  
- CC-cube algorithm, 78
- Cholesky factorization, 4
- communication cost, 95
- communication pipelining, 78
- communication time, 95
- compressible flow, 161
- computation time, 95
- conjugate gradient method, 107
- Connection Machine CM-200, 20
- consecutive execution scheme, 42
- convection-diffusion problem, 178
- corrector step, 36
- coupled two-term recurrence, 66, 92
- Cray CS6400, 160
- Cray J90, 145
- Cray T3E, 5, 35, 160, 225
- cut-through routing, 91
- cyclic distribution, 40
  
- DEC AlphaServer 8400, 146
- degree-4, 78
- diagonal-implicitly iterated RK, 34
- diagonally implicit RK methods, 34
- direct methods, 142
- dirichlet/neumann boundary conditions, 106
- distributed system, 130
- division, 80
- domain decomposition method, 105, 178, 212
- duality methods, 194
- dynamic load balancing, 122
  
- electromagnetic scattering, 215
- error control, 34
- euler equations, 161
- exchange, 80
- extrapolation methods, 34
  
- fill in, 142
- finite difference time domain, 217
- finite differences method, 105
- finite element techniques, 192
- finite elements method, 105, 203
- finite volume time domain, 217
- fixed point techniques, 194
- flit, 95
- forward elimination, 13
- fractional method, 205
- Fujitsu AP3000 multiprocessor, 192
  
- galerkin formulation, 206
- gaussian elimination, 21, 40
- global accumulation, 96
- global communication, 64, 90
- global reduction, 35
- global round robin, 122
- global synchronization points, 64
- global task count, 131
- grid generation algorithm, 163

- grid topologies, 161
- group execution scheme, 42
- hypercube, 91
- hypercube interconnection, 77
- IBM SP Power 2, 106
- implicit methods, 34
- implicit Riemann solver, 161
- implicit Runge-Kutta methods, 34
- inner product, 90
- inner products, 64
- Intel Paragon, 21
- iterative methods, 142
- krylov subspace methods, 64
- lanczos process, 90
- LAPACK, 23
- last transition, 80
- lid-driven cavity flow, 208
- LINPACK, 147
- LINUX, 207
- load balance, 19, 40
- load balancing, 171, 219
- look-ahead technique, 67
- LU factorization, 19
- lubricant fluid, 192
- macrostep, 36
- magnetic field integral equation, 219
- matrix multiplication, 20
- matrix-by-vector products, 64
- matrix-vector multiplications, 90
- mesh, 91
- message passing interface, 43
- message-passing, 35
- MIMD systems, 215
- minimum degree algorithms, 143
- motion equations, 206
- MPI, 204, 224
- multifrontal, 4
- multilevel nested dissection algorithm, 4
- multiple minimum degree algorithm, 5
- multiplicative, 55
- multiplicative algorithm, 54
- navier-stokes equations, 203
- navier-stokes incompressible flow, 182
- NEC SX-4, 145
- nested dissection algorithms, 143
- newton method, 37
- non-Hermitian linear systems, 64
- one-sided Jacobi method, 77
- optimal strategy, 56
- optimized order 2 method, 179
- ordinary differential equations, 34
- overlapping, 98
- overlapping mesh technique, 160
- overlapping subdomains, 108, 161
- parallel direct solvers, 4
- parallel distributed memory, 90
- parallel linear algebra software libraries, 20
- parallel variable distribution, 54
- partial differential equations, 34
- performance model, 95
- per-hop time, 95
- per-word time, 95
- Permuted-BR, 78, 83
- Permuted-D4, 85
- pipelining degree, 81
- PoLAPACK, 21
- Poly LAPACK, 21
- polyalgorithms, 21
- preconditioner, 106
- preconditioning, 223
- perfect matching layers, 217
- pressure poisson equation, 206
- primitive variable formulation, 203
- PSPASES, 5
- PVM, 161, 224
- quasi-minimal residual, 90
- random polling, 122
- ring, 91
- rosenbrock-type methods, 34
- scalability, 5
- ScaLAPACK, 21
- schwartz algorithm, 108
- selection strategy, 56
- semiconductor device simulation, 142
- semiconductor process simulation, 142
- separator, 5
- SGI Origin 2000, 5, 146
- shared memory multiprocessors, 142
- single node accumulation, 95
- single-node broadcast, 95
- single-transfer, 35
- space decomposition, 55
- sparse linear systems, 90
- sparse Linear Systems, 4
- spatial decomposition, 54
- spatial discretization, 34
- s*-stage Radau Ia method, 36
- s*-stage Radau IIa method, 36
- stage vectors, 36
- stage-value, 35
- start-up time, 95
- separator tree, 8
- stepsize selection, 34
- stiff initial value problems, 34

- store-and-forward routing, 91
- strongly convex, 54
- subtree-to-subcube mapping, 8
- SUN Enterprise 3000, 106
- Sun Enterprise 4000, 146
- SUN SPARCstation 20, 106
- supernodal elimination tree, 8
- supernodes, 144
- symbolic factorization, 12, 144
- symmetric eigenvalue, 77
- symmetric positive definite, 4
- termination, 130
- three-term procedure, 66
- time step, 36
- token passing, 132
- two-term recurrences, 90
- uniform, 163
- unsymmetric Lanczos process, 91
- velocity formulation, 203
- wrap-around connection, 95