

**Design and Evaluation of Tridiagonal
Solvers for Vector and Parallel
Computers**

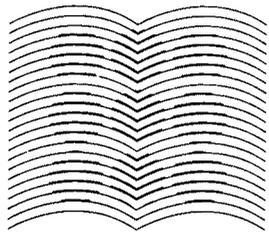
Author: Josep Lluís Llariba Pey
Advisor: Juan José Navarro Guerrero

Barcelona, January 1995



Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

UNIVERSITAT
POLITÈCNICA
DE CATALUNYA



BIBLIOTECA
EX - LIBRIS

Tesi doctoral presentada per Josep Lluís Larriba Pey per tal
d'aconseguir el grau de Doctor en Informàtica per la
Universitat Politècnica de Catalunya

UNIVERSITAT POLITÈCNICA DE CATALUNYA
ADMINISTRACIÓ D'ASSUMPTES ACADÈMICS

Aquesta Tesi ha estat enregistrada
a la pàgina 72 amb el número 658

Barcelona, 31-3-95

L'ENCARREGAT DEL REGISTRE,



UNIVERSITAT
POLITÈCNICA
DE CATALUNYA

Barcelona, 10 de Maio de 1995

To my wife Marta

To my parents Elvira and José Luis

“A journey of a thousand miles must
begin with a single step”

Lao-Tse

Acknowledgements

I would like to thank my parents, José Luis and Elvira, for their unconditional help and love to me. I will always be indebted to you. I also want to thank my wife, Marta, for being the sparkle in my life, for her support and for understanding my (good) moods.

I thank Juanjo, my advisor, for his friendship, patience and good advice. Also, I want to thank Àngel Jorba for his unconditional collaboration, work and support in some of the contributions of this work.

I thank the members of “Comissió de Doctorat del DAC” for their comments and specially Miguel Valero for his suggestions on the topics of chapter 4.

I thank Mateo Valero for his friendship and always good advice.

I thank the “Departament d’Arquitectura de Computadors” and its members for their support and specially Oriol Roig for programming some algorithms.

I thank “CEPBA” for the support it gave to this thesis and, in general, for the support it is giving to basic research.

I thank Dr. Michael Mascagni for his work on part of chapter 6.

I want to thank the “Departamento de Arquitectura y Tecnología de Computadores y Electrónica” of Universidad de Málaga for allowing me to use the Paramid 16/860SYS and specially to Guillermo Pérez Trabado for his patience.

I want to thank the “Comisión Asesora de Investigación Científica y Técnica” for its economic support through projects TIC-299/89, TIC-880/92 and for the “Formación de Personal Investigador” grant I had from 1989 to 1991.

Finally I thank the European Commission for its economic support through ESPRIT basic Research Action 6634 APPARC.

Index

Preface	7
Chapter 1: Setting of the problem	9
1.1 Introduction	10
1.2 Preliminaries	10
1.2.1 Parallelism	15
Cyclic Reduction and R-Cyclic Reduction	16
Strict diagonal dominance for CR	18
Divide and Conquer	19
Strict diagonal dominance for DC	20
1.3 A historic approach to bidiagonal and tridiagonal solvers	21
R-Cyclic Reduction for tridiagonal systems	21
Recursive Doubling	25
Divide and Conquer	26
Bidiagonal systems solvers	27
Other algorithms and methods for tridiagonal systems	28
1.4 A brief summary of applications	29
Alternating Direction Implicit	29
Natural Cubic Splines and B-Splines Curve Fitting	29
Computation of photon statistics in lasers	30
Solution of neuron models by domain decomposition	31

Chapter 2: A unified view of R-Cyclic Reduction and Divide and Conquer	33
2.1 Introduction	34
2.2 Unified view of R-CR and DC	34
2.3 Early termination of the methods	37
2.3.1 Proofs for the early termination criteria	38
2.3.2 Analysis of the early termination of the unified algorithm	41
Error versus diagonal dominance for the unified algorithm	41
Amount of work	42
2.4 Parallel and Vector versions of the unified algorithm	43
Vector version of the unified algorithm	47
Chapter 3: The Overlapped Partitions Method	49
3.1 Introduction	50
3.2 The Overlapped Partitions Method	50
3.3 Proof for the accuracy of OPM	52
3.4 Parallel and Vector versions of OPM	53
3.5 Other work related to OPM	55
Chapter 4: Bidiagonal solvers on vector processors	57
4.1 Introduction	58
4.2 Previous work	58
4.3 Target architecture	59
Modelling the execution time	60
The Convex C-3480	62
4.4 Vector algorithms	62
4.4.1 R-Cyclic Reduction	63
Analysis of the assembly vector code of R-Cyclic Reduction	64

Scheduling of R-CR for vector processors	65
A general model for R-CR	67
Analyzing the model for different computers	67
4.4.2 Divide and Conquer	69
Analysis of the assembly vector code of Divide and Conquer	71
Scheduling of DC for vector processors	72
A general model for Divide and Conquer	73
Finding an optimum value of the partial unrolling U	76
Remarks on finding the optimum number of partitions on a real computer	77
Analyzing the model for different computers	78
4.4.3 The Overlapped Partitions Method	81
Analysis of the assembly vector code of the Overlapped Partitions Method	81
Scheduling of OPM for vector computers	82
A general model for OPM	82
Finding an optimum value of the partial unrolling U	83
Optimum number of partitions for OPM	84
Analyzing the model for different computers	84
4.5 Comparison of the models for different types of computers	85
Chapter 5: Bidiagonal solvers on parallel computers	91
5.1 Introduction	92
5.2 Previous work	92
5.3 Target architecture	92
Modelling the execution time of a parallel algorithm	94
5.4 The parallel algorithms	95
5.4.1 Divide and Conquer	95
A model for DC	96

5.4.2	R-Cyclic Reduction	97
	A model for the two step R-Cyclic Reduction	101
5.4.3	The Overlapped Partitions Method	102
	A model for OPM	103
5.5	Comparison of the methods on different types of architectures	104
	An example on the Paramid 16/860SYS	110
	The Paramid 16/860SYS	110
	PVM on the Paramid 16/860SYS	111
	Implementation of DC on the Paramid 16/860SYS	112
Chapter 6:	Applications	113
6.1	Introduction	114
6.2	Natural Cubic Splines and B-Spline curve fitting	114
6.2.1	A method based on the TJ decomposition	115
6.2.2	An early termination of the TJ decomposition	117
6.2.3	Description and analysis of the method by Chung and Shen	118
6.2.4	Comparison of the methods	119
6.3	Solution of neuron models by domain decomposition techniques	120
6.3.1	The domain decomposition method	121
6.3.2	Analysis of the discretization matrices	125
6.3.3	Phases 1 and 3	125
6.3.4	Phase 2	127
6.3.5	Proofs for formulae (6.4) and (6.5)	127

Conclusions and future work131
Theoretical aspects of the contributions131
Practical aspects of the contributions132
Future work136
Appendix137
1 Basic LU-type decompositions138
LU decomposition for general systems138
LDU decomposition139
LDLT decomposition for symmetric systems140
Cholesky factorization140
Taha and Liaw variant of Evans decomposition for Toeplitz systems141
TJ decomposition for strictly diagonal dominant systems143
Malcolm and Palmer decomposition for Toeplitz symmetric systems143
2 Comparison of the LU-type decompositions144
References147

Preface

The solution of tridiagonal systems on parallel and vector computers has been studied and analyzed widely in the last two decades. The reasons for this interest are twofold. From the academic point of view, the problem is interesting as it is a challenge because it shows very little inherent parallelism and, at the same time, it is simple and easy to be understood. From the engineering point of view it is also interesting because the applications in which tridiagonal systems have to be solved come from very diverse fields. Examples of these applications are the use of the ADI iterative solver for partial differential equations, the computation of Natural Cubic Splines and B-Splines, the multidimensional diffusion computations, the computation of photon statistics in lasers or the discretization of neuron models by domain decomposition, etc.

Some of the applications mentioned above require the solution of several tridiagonal systems at different moments with the same coefficient matrix. For this reason, it is convenient to find an LU decomposition of the matrix which gives rise to the solution of two bidiagonal systems every time that a tridiagonal system has to be solved anew. So, here we concentrate on the solution of the bidiagonal systems that arise from LU -type decompositions of tridiagonal systems.

In this thesis we intend to cover the theoretical and practical aspects of the study of parallel and vector bidiagonal systems solvers. From the theoretical point of view, we unify the algorithms of the most popular methods for the solution of bidiagonal systems. Also, for those methods we unify the formulae that allow to determine their early termination for the solution of strictly diagonal dominant solvers. Additionally, we propose the Overlapped Partitions Method which is suited for the solution of strictly diagonal dominant bidiagonal systems. From the practical point of view, we analyze the methods for vector and parallel computers. We study computational models and practical implementations of the methods and analyze two different applications.

In order to cover all the aspects of the problem, we have structured this work into six chapters, the sum up of some conclusions and an appendix. In chapter one we set the problem in its context. In order to do so, we give some basic definitions that help the reader to understand the type of bidiagonal systems analyzed here. Then, we give a historic view of the evolution of parallel tridiagonal and bidiagonal systems solvers. Finally, we overview some of the applications that give rise to the solution of tridiagonal systems.

In the second chapter we propose a unification of the most popular methods for the solution of bidiagonal systems. These methods are the R -Cyclic Reduction algorithm (R -CR) and the Divide and Conquer algorithm (DC) which have always been regarded as two completely different methods. The methods are unified from the point of view of the algorithm and from the point of view of the early termination criteria for strictly diagonal dominant systems. Also in this chapter, we describe the parallel and vector characteristics of the unified algorithm.

In the third chapter we propose a new method called the Overlapped Partitions Method (OPM) which is a parallel solver for strictly diagonal dominant bidiagonal systems. Here, we describe OPM and prove its accuracy for bidiagonal systems.

Chapter four is devoted to the analysis of R -CR, DC and OPM on five different types of vector processors. Each of the methods is analyzed to optimize its execution on such computers. On one side, the characteristics of each algorithm are studied in order to minimize its traffic with memory. On the other side, the algorithms are analyzed in order to find optimum schedulings for each of the five architectures studied. Models are built and validated with the help of executions on the Convex C-3480. Finally, a global comparison of the methods is performed in order to determine the fastest method given the characteristics of the problem to be solved, i. e. diagonal dominance, order and maximum error allowed to the solution.

In chapter five we analyze the methods on parallel computers. We study the influence of the relation between the computation and communication times on the global execution time of the algorithms. With this aim in mind, we study DC and OPM and propose a variation of R -CR. Finally, we compare the methods on three modelled architectures. The three architectures have different values of the computation time, the start-up time for a communication and the time to send an element. Finally, we give an example of the execution of DC on the Paramid 16/860SYS.

In chapter six we analyze the tridiagonal systems that appear in two different applications: curve fitting by Natural Cubic Splines and B-Splines and the solution of the models of the electric behaviour of neurons by domain decomposition. For the former application, we propose the early termination of a method based on an LU -type decomposition. For the latter application, we propose some strategies to save work to the solution of the problem by domain decomposition.

The conclusions and future intended work are presented in a chapter. There, we sum up the most important contributions of this work and generalize the results obtained for the different architectures studied. Finally, we explain the intended research lines spinned off from some of the topics studied in this work.

The appendix is devoted to the description and comparison of the different LU -type decompositions for tridiagonal systems that have been proposed in the literature. Those decompositions are compared in terms of amount of work they generate.

Chapter 1: Setting of the problem

In this chapter we describe the context of the problem. First, we give a few preliminaries that help the reader to understand the type of bidiagonal systems that are analyzed and where they come from. Then, we give a historic view of the evolution of the methods for the parallel solution of tridiagonal and bidiagonal systems. After that, we overview some real applications in which the tridiagonal systems that have to be solved require their decomposition into a set of bidiagonal systems.

1.1 Introduction

This thesis focusses on the parallel solution of bidiagonal systems that arise from the decomposition of tridiagonal matrices into L and U factors. The problem is analyzed both from the point of view of the theoretical issues of the existing and new methods and from the point of view of the practical implementations of those methods on vector and parallel architectures.

In the following section we give some basic preliminary definitions that help the reader to understand the type of problem dealt with in the rest of the work. We describe the tridiagonal systems that we want to solve and the basic scalar algorithm to solve them. Then, we describe the classic LU decomposition and name other LU -type decompositions used to factorize tridiagonal matrices. Finally, we define the type of bidiagonal system that we want to solve in parallel and describe the most popular parallel methods.

Most of the parallel bidiagonal solvers that we analyze in this thesis emerge from the solvers designed for the parallel solution of tridiagonal systems. For this reason, section 1.3 is devoted to give a historic account of the different methods for the solution of tridiagonal, block tridiagonal, bidiagonal and narrow banded systems in general. Those methods are basically the R -Cyclic Reduction, the Recursive Doubling and the Divide and Conquer families of algorithms. With this historic approach we intend to set the appropriate context for the unification of R -Cyclic Reduction and Divide and Conquer that we propose in chapter 2.

Finally, in section 1.4 we overview different applications in which tridiagonal systems have to be solved. Some of those tridiagonal systems have to be solved repeated times and for this reason an LU -type decomposition is recommended in order to save some work during the solution of those systems. Those applications come from very diverse fields of the computational sciences as we said in the introduction.

1.2 Preliminaries

The solution of tridiagonal systems of equations $Ax = b$

$$(1.1) \quad Ax = \begin{bmatrix} d_{[1]} & c_{[1]} & & & 0 \\ e_{[2]} & d_{[2]} & c_{[2]} & & \\ & e_{[3]} & d_{[3]} & c_{[3]} & \\ & & \dots & \dots & \dots \\ 0 & & & e_{[N]} & d_{[N]} \end{bmatrix} \begin{bmatrix} x_{[1]} \\ x_{[2]} \\ x_{[3]} \\ \dots \\ x_{[N]} \end{bmatrix} = \begin{bmatrix} b_{[1]} \\ b_{[2]} \\ b_{[3]} \\ \dots \\ b_{[N]} \end{bmatrix}$$

of order N can be performed with Gaussian elimination which is a purely sequential algorithm. This algorithm is shown in Figure 1.1. There, one starts by zeroing element $e_{[2]}$ down to element $e_{[N]}$ during the forward elimination and element $c_{[N-1]}$ up to element $c_{[1]}$ during the backsubstitution. The arithmetic operation count of this strategy is $3(N-1)$ subtractions, $3(N-1)$ products and $2N-1$

divisions. However, the algorithm can be transformed to change the $2N - 1$ divisions into $N - 1$ divisions and $2(N - 1)$ products.

	Subtractions	Products	Divisions
Gaussian elim.	$3(N - 1)$	$3(N - 1)$	$2N - 1$
Transformed Gaussian elim.	$3(N - 1)$	$5(N - 1)$	$N - 1$

Table 1.1. Operation count for Gaussian elimination on a tridiagonal system.

We define the diagonal dominance of (1.1) by $\delta = \min_i (|d_{[i]}| / |e_{[i]} + c_{[i]}|)$. We say that a system is diagonal dominant if $\delta \geq 1$ and strictly diagonal dominant if $\delta > 1$.

```

do i = 2, N
    aux = e[i] / d[i-1]
    d[i] = d[i] - c[i-1] aux
    b[i] = b[i] - b[i-1] aux
enddo
x[N] = b[N] / d[N]
do i = N - 1, 1, -1
    x[i] = (b[i] - x[i+1] c[i]) / d[i]
enddo

```

Figure 1.1. Gaussian elimination for tridiagonal systems of equations.

Gaussian elimination is stable for diagonal dominant and strictly diagonal dominant systems. In this thesis we consider this type of systems that, in addition, appear in different applications.

Gaussian elimination may not be stable if the system is not diagonal dominant or strictly diagonal dominant. A stable approach for this case may be the use of Givens rotations to perform the elimination of the elements. The use of this strategy, though, implies that more operations are performed. It is not advisable to use Givens rotations for diagonal dominant and strictly diagonal dominant systems.

If several tridiagonal systems have to be solved with the same coefficient matrix at different moments it

is convenient to perform a decomposition of matrix A into an upper and lower bidiagonal multiplicative factors $A = LU$

$$L = \begin{bmatrix} 1 & & & & & \\ l_{[2]} & 1 & & & & \\ & l_{[3]} & 1 & & & \\ & & & \dots & \dots & \\ & & & & l_{[N]} & 1 \end{bmatrix} \quad U = \begin{bmatrix} u_{[1]} & c_{[1]} & & & & \\ & u_{[2]} & c_{[2]} & & & \\ & & u_{[3]} & \dots & & \\ & & & & \dots & c_{[N-1]} \\ & & & & & u_{[N]} \end{bmatrix}$$

One algorithm to compute such decomposition is shown in Figure 1.2 and its arithmetic operation count is $N - 1$ subtractions, $N - 1$ products and $N - 1$ divisions.

```

u[1] = d[1]
do i = 2, N
    l[i] = e[i] / u[i-1]
    u[i] = d[i] - c[i-1] l[i]
enddo

```

Figure 1.2. LU decomposition of a tridiagonal matrix.

	Subtractions	Products	Divisions
LU decomposition	$N - 1$	$N - 1$	$N - 1$

Table 1.2. Operation count for the computation of the LU decomposition of a tridiagonal system.

With this decomposition two bidiagonal systems $Ly = b$ and $Ux = y$ have to be solved to find the solution to the original system $Ax = b$. This decomposition has to be performed only once independently of the number of times the original system has to be solved with different right hand side vectors.

The solution to the lower $Ly = b$ and upper $Ux = y$ bidiagonal systems that arise from this decomposition implies a lower operation count than the straight solution of the tridiagonal system. The algorithm for the solution of those bidiagonal systems is shown in Figure 1.3. The arithmetic operations used with this strategy add up to $2(N - 1)$ subtractions, $2(N - 1)$ products and $N - 1$ divisions for each new system to be solved as summarized in Table 1.3.

Variations of this LU decomposition lead to algorithms without divisions during the solution of $Ly = b$ and $Ux = y$. Those variations are explained in the appendix of this document.

```

do i = 2, N
    y[i] = b[i] - y[i-1] l[i]
enddo
x[N] = y[N] / u[N]
do i = N-1, 1, -1
    x[i] = (y[i] - x[i+1] c[i]) / u[i]
enddo
    
```

Figure 1.3. Solution of $Ly = b$ and $Ux = y$.

	Subtractions	Products	Divisions
$Ly = b$ and $Ux = y$	$2(N-1)$	$2(N-1)$	$N-1$

Table 1.3. Operation count for computing $Ly = b$ and $Ux = y$.

If it is necessary to solve K tridiagonal systems at different moments with the same coefficient matrix, the use Gaussian elimination implies $3K(N-1)$ subtractions, $3K(N-1)$ products and $2KN-K$ divisions. On the contrary, if the LU decomposition is first performed and then $Ly = b$ and $Ux = y$ are solved for each new system, we perform a start-up of $N-1$ subtractions, $N-1$ products and $N-1$ divisions and then for each new system we execute $2(N-1)$ subtractions, $2(N-1)$ products and $N-1$ divisions. The bulk number of equations for the solution of a tridiagonal system with both strategies is shown in Table 1.4. There, it is possible to see that Gaussian elimination is in clear disadvantage.

	Gaussian elim.	$LU, Ly = b$ and $Ux = y$
$K = 1$	$8N-7$	$8(N-1)$
$K = 2$	$16N-14$	$13(N-1)$
K	$K(8N-7)$	$3(N-1) + 5K(N-1)$

Table 1.4. Operation count for computing the solution to K systems.

Although we have used the classic LU decomposition up to here, there are different decompositions for different types of tridiagonal matrices. For non-Toeplitz non-symmetric matrices (also called general here) one can use the classic LU decomposition or variations of it. For symmetric matrices one can use the LDL^T or the Cholesky factorizations. For Toeplitz matrices there are decompositions like that by Evans [Evan80] that was revisited by Taha and Liaw [TaLi93] which give rise to Toeplitz factors. Finally, for symmetric strictly diagonal dominant Toeplitz matrices there is a modification of the classic LU decomposition proposed by Malcolm and Palmer [MaPa74] that leads to almost Toeplitz factors. The LU -type decompositions mentioned here are described and analyzed in detail in the appendix of this thesis.

In this thesis we concentrate on the solution of the following type of systems

$$(1.2) \quad Ax = \begin{bmatrix} 1 & & & & 0 \\ a_{[2]} & 1 & & & \\ & a_{[3]} & 1 & & \\ & & \dots & \dots & \\ 0 & & & a_{[N]} & 1 \end{bmatrix} \begin{bmatrix} x_{[1]} \\ x_{[2]} \\ x_{[3]} \\ \dots \\ x_{[N]} \end{bmatrix} = \begin{bmatrix} b_{[1]} \\ b_{[2]} \\ b_{[3]} \\ \dots \\ b_{[N]} \end{bmatrix}.$$

of which $Ly = b$ is an example. Also, system $Ux = y$ can be transformed to be of the type of (1.2) by changing the operations of the decomposition and the solution of the system as shown in Figure 1.4. This, changes the arithmetic operation count of the decomposition into $N - 1$ subtractions, $3(N - 1)$ products and N divisions and the arithmetic operation count of the solution of each tridiagonal system into $2(N - 1)$ additions and $3(N - 1)$ products. This means that the operation count of the decomposition has been incremented but the operation count of the solution has been maintained regardless of the type of operation. The bulk of operations for the solution of K systems is shown in Table 1.6. In this case, the operation count of the modification of Figure 1.4 is higher than for the LU decomposition described previously but it is important to say that the number of divisions per equation has been reduced considerably.

Decomposition	Solution
$u_{[1]} = 1/d_{[1]}$	do $i = 2, N$
do $i = 2, N$	$y_{[i]} = b_{[i]} - y_{[i-1]}l_{[i]}$
$l_{[i]} = e_{[i]}u_{[i-1]}$	enddo
$u_{[i]} = 1/(d_{[i]} - c_{[i-1]}l_{[i]})$	do $i = N, 1, -1$
$c_{[i-1]} = c_{[i-1]}u_{[i-1]}$	$y_{[i]} = y_{[i]}u_{[i]}$
enddo	enddo
	do $i = N - 1, 1, -1$
	$x_{[i]} = y_{[i]} - x_{[i+1]}c_{[i]}$
	enddo

Figure 1.4. Solution of $Ly = b$ and $Ux = y$.

As a consequence, we suppose that the type of systems that we are going to solve are those like (1.2). We define the diagonal dominance of (1.2) by $\delta = \min_i (1/|a_{[i]}|)$.

	Substractions	Products	Divisions
<i>LU</i> decomposition	$N - 1$	$3(N - 1)$	N
$Ly = b$ and $Ux = y$	$2(N - 1)$	$3(N - 1)$	-

Table 1.5. Operation count for computing the *LU* decomposition and solution of $Ly = b$ and $Ux = y$ of Figure 1.4.

	Gaussian elim.	<i>LU</i> , $Ly = b$ and $Ux = y$	<i>LU</i> modified
$K = 1$	$8N - 7$	$8(N - 1)$	$10N - 9$
$K = 2$	$16N - 14$	$13(N - 1)$	$15N - 14$
K	$K(8N - 7)$ with $2KN - K$	$3(N - 1) + 5K(N - 1)$ with $(K + 1)(N - 1)$ divisions	$5N - 4 + 5K(N - 1)$ with N divisions

Table 1.6. Operation count for computing K tridiagonal systems.

1.2.1 Parallelism

Gaussian elimination applied to (1.1) and (1.2) has no parallelism at the iteration level because in order to compute iterate i it is necessary to compute iterate $i-1$ before. Thus the parallel solution of (1.1) and (1.2) requires different algorithms. Those algorithms are based on manipulations of the arithmetic expressions of the algorithms of Figures 1.1 and 1.4. Those manipulations take profit of the commutative and associative properties of the addition and product operations. Nevertheless, those manipulations imply more work than for Gaussian elimination in order to perform parallelism.

One possible approach is to apply eliminations of elements in an order which is different from that used by Gaussian elimination. In general, this different approach implies that while an element is eliminated, some fill-in is introduced. For instance, in the matrix shown in Figure 1.5 we show the fill-in that results

from the elimination of element $e_{[n]}$ without having eliminated element $e_{[n-1]}$ previously. Note that the fill-in introduced has to be eliminated in some manner. This means that parallelism can be achieved because different elements $e_{[i]}$ can be eliminated at the same time without having eliminated elements $e_{[i-1]}$ beforehand at the cost of performing a larger number of arithmetic operations than that performed by the sequential algorithm. In general, if the parallelism offered by the target architecture compensates for the increment of operations, the parallel algorithm is faster than the sequential one.

The parallel methods that we are going to analyze in this thesis are based in the strategy above mentioned. In the following paragraphs we describe Cyclic Reduction, R -Cyclic Reduction and Divide and Conquer that are the methods we will deal with in this thesis.

$$\begin{bmatrix} \dots & \dots & & & \\ e_{[n-1]} & 1 & & & \\ & e_{[n]} & 1 & & \\ & & & \dots & \dots \end{bmatrix} \begin{bmatrix} \dots \\ x_{[n-1]} \\ x_{[n]} \\ \dots \end{bmatrix} = \begin{bmatrix} \dots \\ b_{[n-1]} \\ b_{[n]} \\ \dots \end{bmatrix} \rightarrow \begin{bmatrix} \dots & \dots & & & \\ e_{[n-1]} & 1 & & & \\ e'_{[n]} & 0 & 1 & & \\ & & & \dots & \dots \end{bmatrix} \begin{bmatrix} \dots \\ x_{[n-1]} \\ x'_{[n]} \\ \dots \end{bmatrix} = \begin{bmatrix} \dots \\ b_{[n-1]} \\ b'_{[n]} \\ \dots \end{bmatrix}$$

Figure 1.5. Gaussian elimination basic operation on a general element of the subdiagonal of a bidiagonal system.

Cyclic Reduction and R -Cyclic Reduction

The description of Cyclic Reduction for tridiagonal systems can be found in many different general books as a didactic element to explain parallelism in one of its forms [HoJe81, HoJe88, Sch87, BeTs89]. Now, we reproduce the bidiagonal version of the description of Cyclic Reduction given in [HoJe81, HoJe88] and then we deduce R -CR.

For the description of CR, we suppose that the order of the system to be solved is $N = 2^q$ for simplicity. Now, let us suppose that we have two adjacent equations of a first order recurrence for $i = 2, 4, \dots, N$

$$(1.3) \quad \begin{array}{rcl} a_{[i-1]} x_{[i-2]} + & x_{[i-1]} & = b_{[i-1]} \\ & a_{[i]} x_{[i-1]} + x_{[i]} & = b_{[i]} \end{array}$$

By multiplying equation $i - 1$ by $a_{[i]}$ and subtracting it from equation i , we have that variable $x_{[i-1]}$ can be eliminated from equation i and we have

$$(1.4) \quad a_{[i]}^{(1)} x_{[i-2]}^{(1)} + x_{[i]}^{(1)} = b_{[i]}^{(1)}$$

From the description of CR, it is easy to obtain a description of R -CR. In this case, we have to eliminate $R - 1$ equations out of R supposing that $N = R^q$. Also, we have R adjacent equations for $i = R, 2R, \dots, N$

$$(1.6) \quad \begin{array}{ccccccc} a_{[i-R+1]}x_{[i-R]} + & x_{[i-R+1]} & & & & & = b_{[i-R+1]} \\ & a_{[i-R+2]}x_{[i-R+1]} + & x_{[i-R+2]} & & & & = b_{[i-R+2]} \\ & & \dots & & & & \dots \\ & & & a_{[i]}x_{[i-1]} + & x_{[i]} & & = b_{[i]} \end{array}$$

A similar procedure to that performed to (1.3) eliminates variables $x_{[i-R+1]}$ through $x_{[i-1]}$ leading to the following equation after one level of reduction

$$(1.7) \quad a_{[i]}^{(1)}x_{[i-R]}^{(1)} + x_{[i]}^{(1)} = b_{[i]}^{(1)}$$

This can be performed in parallel to sets of R equations at each level of reduction as it was done for CR. The process can be repeated recursively for $S = \log_R N$ levels of reduction.

Strict diagonal dominance for CR

In the case of strict diagonal dominance $\delta > 1$, the off-diagonal elements of the matrix decrease in absolute value compared to the main diagonal elements at each level of reduction for the case of Cyclic Reduction [HoJe88]. This means that at a certain reduction level it is possible to drop the off-diagonal elements of the matrix as they can be considered to be zero. Thus we obtain the identity matrix, the system can be solved trivially and the backward procedure can start. This is called early termination of CR.

We illustrate this behaviour of the off-diagonal elements of the matrix with a simple example in Figure 1.7. There, we can see that the value of the subdiagonal elements after each level of reduction keep decreasing. The diagonal dominance δ of the example system is 2 originally and keeps growing to 4 and 8 after the first and second levels of reduction.

$$\begin{bmatrix} 1 & & & & & & \\ 0.5 & 1 & & & & & \\ & 0.5 & 1 & & & & \\ & & 0.5 & 1 & & & \\ & & & 0.5 & 1 & & \\ & & & & 0.5 & 1 & \\ & & & & & 0.5 & 1 \\ & & & & & & 0.5 \dots \end{bmatrix}^{(0)} \rightarrow \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & 0.5 & 1 & & & & \\ & -0.25 & & 1 & & & \\ & & 0.5 & & 1 & & \\ & & & -0.25 & & 1 & \\ & & & & 0.5 & & 1 \\ & & & & & -0.25 & 1 \\ & & & & & & 0.5 \dots \end{bmatrix}^{(1)} \rightarrow \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & 0.5 & 1 & & & & \\ & & & 1 & & & \\ & & & 0.5 & 1 & & \\ & & & -0.25 & & 1 & \\ & & & & 0.5 & & 1 \\ & & & & & 0.0625 & 1 \\ & & & & & & 0.5 \dots \end{bmatrix}^{(2)}$$

Figure 1.7. Growth of diagonal dominance for 3 reductions of CR.

Divide and Conquer

Divide and Conquer has been described and analyzed in many different papers. Its description, though, has been almost always based on the partitioning of the system into a set of blocks that are transformed during the algorithm. We base this description on the partitioning of the system into blocks.

We want to solve system $Ax = b$ where A is the lower bidiagonal matrix defined in chapter 1. Here, we define element i of vector z as $z_{[i]}$ and element i of subvector z_j as $z_{j[i]}$. Also, we define element $[i,j]$ of matrix Z as $Z_{[i,j]}$ and element $[i,j]$ of submatrix Z_k as $Z_{k[i,j]}$.

Let us suppose that N , the order of the system, is a power of an arbitrary number R for simplicity. Then, matrix A is partitioned into P -by- P blocks of size R -by- R (where $P = N/R$) as shown in Figure 1.8. Also, vectors x and b are partitioned correspondingly into P subvectors of size R that can be named x_p and b_p for $1 \leq p \leq P$. Note that only element $[1,R]$ of submatrices L_p for $1 < p \leq P$ is different from zero. We define column R of submatrix L_p as vector l_p .

$$A = \begin{bmatrix} 1 & & & 0 \\ a_{[2]} & 1 & & \\ & \dots & \dots & \\ 0 & & a_{[N]} & 1 \end{bmatrix} = \begin{bmatrix} D_1 & & & 0 \\ L_2 & D_2 & & \\ & \dots & \dots & \\ 0 & & L_P & D_P \end{bmatrix} \quad D_p = \begin{bmatrix} 1 & 0 & 0 \\ a_{[i+2]} & 1 & 0 \\ 0 & \dots & \dots & 0 \\ 0 & 0 & a_{[Rp]} & 1 \end{bmatrix}$$

$$L_p = \begin{bmatrix} 0 & \dots & a_{[i+1]} \\ 0 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & 0 \end{bmatrix} \quad l_p = \begin{bmatrix} a_{[i+1]} \\ 0 \\ \dots \\ 0 \end{bmatrix} \quad \begin{aligned} i &= R(p-1) \quad , 1 \leq p \leq P \\ L_{p[1,R]} &= l_p[1] = a_{[R(p-1)+1]} \quad , 1 < p \leq P \\ L_{p[j,R]} &= l_p[j] = 0 \quad , 2 \leq j \leq R \end{aligned}$$

Figure 1.8. Matrix A from $Ax = b$. Submatrices D_p and L_p . Reduction of submatrix L_p to vector l_p .

Based on this partitioning, the Divide and Conquer algorithm can be described as follows:

Forward phase. During this phase, each submatrix D_p is diagonalized. With this, fill-in is introduced in all the elements of vectors l_p . These new filled vectors can be called t_p . This is shown in Figure 1.9 for the case $N = 12$ and $R = 4$. Also, the right hand side vectors b_p are modified. These modified vectors can be called g_p .

After this phase, the system has been decoupled. Only equations in rows Rp keep the recursion (elements in rows 4, 8 and 12 in Figure 2) and form a bidiagonal system of size P . Equations in partition p depend only on equation $R(p-1)$, except for partition 1 in which all equations have already been solved. This is shown in Figure 1.9.

With the notation given above, this phase is equivalent to solve $D_1 g_1 = b_1$ and $D_p \{t_p, g_p\} = \{l_p, b_p\}$ for $1 < p \leq P$.

Solution of the reduced system. Matrix $A^{(1)}$ and vector $b^{(1)}$ are formed by gathering rows R_p of matrix A and vector b for $1 \leq p \leq P$. We call $A^{(1)} x^{(1)} = b^{(1)}$ reduced system. This system is of size P and is solved in this phase with the help of any bidiagonal system solver, i.e. Gaussian elimination, Recursive Doubling, R -CR, etc. $x^{(1)}$ is the vector that contains the solutions to elements $x_{[Rp]}$ of the global solution vector x , that is, $x_{[p]}^{(1)} = x_{[Rp]}$ for $1 \leq p \leq P$.

Backward phase. With the help of the solutions to the reduced system, elements R_p of x , find the solution to the rest of equations. This is equivalent to perform the following operation on each partition $x_p = g_p - x_{[p-1]}^{(1)} t_p$ for $2 < p \leq P$.

Divide and Conquer is stable for diagonal dominant and strictly diagonal dominant systems [VaDe89].

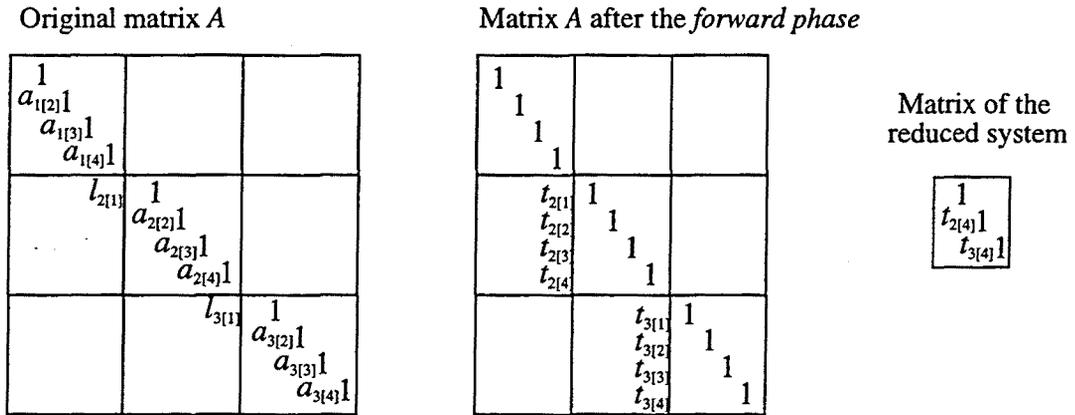


Figure 1.9. Original matrix A, matrix A after the forward phase and matrix of the reduced system during the solution of a bidiagonal system with Divide and Conquer. Example for $N = 12$ and $R = 4$.

Strict diagonal dominance for DC

In the case of strict diagonal dominance, the elements of the solution vector t_p to $D_p t_p = l_p$ for $1 < p \leq P$ decrease in absolute value as they approach to position R $|t_{p[i+1]}| < |t_{p[i]}|$. In this case it is possible to suppose that the off-diagonal elements of the reduced system $A^{(1)} x^{(1)} = b^{(1)}$ are equal to zero. Then, the solution of the reduced system reduces to the solution of the identity system $Ix^{(1)} = b^{(1)}$ and it is not necessary to perform computations to solve it. This is called early termination of DC.

We illustrate this behaviour of the elements of vector t_p with a simple example in Figure 1.10. The example shows one partition of size 6 of the same matrix used in the example for CR in Figure 1.7. There, we can see that after the forward phase, matrices D_p have been diagonalized and vector t_p has been filled. Note that the values of vector t_p decrease in absolute value as they approach position 6.

$$\begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & & & & & \\ 0.5 & 1 & & & & \\ & 0.5 & 1 & & & \\ & & 0.5 & 1 & & \\ & & & 0.5 & 1 & \\ & & & & 0.5 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.5 \\ -0.25 \\ 0.125 \\ -0.0625 \\ 0.03125 \\ -0.015625 \end{bmatrix} \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

Figure 1.10. Matrix D_p and vector l_p after the forward phase.

1.3 A historic approach to bidiagonal and tridiagonal solvers

There is a lot of literature about the parallel solution of tridiagonal systems of equations. Different algorithms and application specific architectures have been proposed to solve such systems in parallel but most of them are based on the work developed during the sixties and seventies.

The first specific method for the solution of tridiagonal systems of equations was the Cyclic Reduction algorithm (also called Odd-Even reduction) proposed by Hockney and Golub in 1965 [Hock65]. This was the precursor of R -Cyclic Reduction, as we saw above, and it was not originally thought as a parallel method. The first parallel tridiagonal solver to be proposed was the Recursive Doubling (RD) algorithm proposed by Stone in 1973 [Ston73]. RD uses transformations based on the associative and commutative properties of the addition and product which are different from those used by CR.

Later, other parallel methods were proposed, namely, a method by Evans and Hatzopoulos proposed in 1975 [EvHa75], two different algorithms proposed by Sameh and Kuck in 1978 [SaKu78] and a method by Swarztrauber proposed in 1979 [Swar79]. The algorithm by Evans and Hatzopoulos is a modification of Gaussian elimination and it is used in the Linpack library to solve tridiagonal systems in scalar mode [DMBS79]. This method has a very limited parallelism. Hatzopoulos proposed a variation of this method based on Givens rotations [Hatz82]. The two algorithms by Sameh and Kuck used Givens rotations and were the precursors of the Divide and Conquer algorithm described above. DC has also been called Spike Algorithm by Sameh and Kuck and, Partition Method by Wang in [Wang81]. The method by Swarztrauber is based on Cramer's rule and a model of the error of this algorithm has been proposed by Tsao [Tsao94].

In the following paragraphs we give a historic perspective of the R -Cyclic Reduction family of methods, the Recursive Doubling algorithm and Divide and Conquer algorithm which are the most popular bidiagonal and tridiagonal solvers. Then, we overview the papers that have dealt with bidiagonal systems and, finally, we mention some parallel methods that do not fall into the categories above mentioned.

R -Cyclic Reduction for tridiagonal systems

The very first ancestor of the R -Cyclic Reduction family of algorithms was Cyclic Reduction (CR), as we said. The motivation of the paper by Hockney was to propose Fourier analysis as a fast direct solver for the solution of Poisson's equation on a rectangular grid. Traditionally, iterative solvers had been used to solve that problem showing to be very slow. In this case, CR dealt with the solution of periodic tridiagonal systems faster than Gaussian elimination on the IBM 7090 and 7094. Nevertheless, in this seminal paper,

Hockney did not refer to the parallel characteristics of the method. The algorithm in [Hock65] had the restriction that the size of the system had to be $N = 2^p - 1$ for any integer value of p . In that paper, Hockney pointed out that under strict diagonal dominance, some of the work of CR could be avoided without perturbing significantly the accuracy of the result. This was called later early termination of CR.

A few authors discussed the use of CR to solve block tridiagonal systems after the first proposal of Cyclic Reduction in 1965. Buzbee, Golub and Nielsen proposed a block tridiagonal implementation of CR in 1970 [BuGN70]. There, one out of every two blocks were decoupled at each step. The blocks were all supposed to be of the same size m and the coefficient matrix had to be formed by $n \times n$ blocks where $n = 2^q - 1$ for any integer value of q . The purpose of the paper was the use of the Block CR algorithm to solve the systems arising from the discretization of Poisson's equation on rectangular and L-shaped grids. In the paper, the authors also described and analyzed a variation of CR attributed to Buneman which changed the original way in which the right hand side of the system was modified along the different steps of CR. Later, in 1974, Swartztrauber extended the Block CR algorithm proposed in [BuGN70] to solve separable elliptic equations [Swar74]. In the same year, Sweet proposed a generalization of the Block Cyclic Reduction algorithm in which $\alpha_i - 1$ out of every α_i blocks were decoupled at each step i of the algorithm [Swee74]. In this case, the blocks were supposed to be of the same size for all partitions at a certain step although the algorithm allowed variations of the elimination factor α_i from step to step. The conclusion of Sweet's paper was that $\alpha_i = 2$ was more advantageous in terms of the amount of arithmetic work than $\alpha_i > 2$ for block tridiagonal systems. Also in 1974, Heller proved that the Odd-Even decoupling of blocks in CR is a particular case of the simultaneous decoupling of all the blocks at each step [Hell74, remark 9, p. 493]. This has important implications on the massive parallel solution of tridiagonal and block tridiagonal systems and was studied later in the bibliography as we will see below. Finally, in 1977, Sweet adapted the Block CR algorithm proposed in [BuGN70] to work on an arbitrary number of partitions n [Swee77].

In 1975 CR was first analyzed as a parallel non-block tridiagonal equation solver by Stone [Ston75] and by Lambiotte and Voigt [LaVo75]. In the paper by Stone, the parallel operation counts of CR, Recursive Doubling and the variation of CR by Buneman for non-block systems were compared. The conclusions of the paper were that the vector operation count for CR is smaller than for the other methods while the parallel operation count on array processors is similar for both CR and Recursive Doubling. In the paper by Lambiotte and Voigt, a theoretical comparison of direct and iterative solvers on a model of the CDC STAR-100 vector computer was made. The direct methods were Gaussian elimination, Recursive Doubling and CR; the iterative methods were the Jacobi iteration, the successive overrelaxation and a method attributed to Traub. The conclusions of the paper showed that CR is the best algorithm of those analyzed for an architecture of the type of the CDC STAR-100. Nevertheless, Gaussian elimination and Recursive Doubling can run faster than CR for systems smaller than 100 on the the same type of architecture. Nevertheless, the most important conclusions of this paper were for Recursive Doubling as we will see.

From the writing of those papers on, CR was mainly analyzed and used on vector computers.

Nevertheless, the interleaved memories of vector computers supposed a great drawback for the use of CR. The problem of CR resides in the fact that the accesses to memory during the different steps of the algorithm are with strides a power of 2 which cause memory bank conflicts on interleaved memories. Several attempts were made to overcome this problem being the first one proposed by Kershaw in 1982 [Kers82]. Kershaw solved the problem of the strides by reordering the data in order to allow only strides equal to 1 on the Cray-1. A similar approach was used by Abu-Shumays in the vectorization of the tridiagonal systems arising from the discretization of multidimensional diffusion computations [Abu-85]. Nevertheless, the approach by Kershaw was improved by Munthe-Kaas in 1989 [Munt89] and by de Groen in 1991 [deGr91]. These authors proposed different variations of the generalized CR proposed by Sweet in [Sweet74]. On one hand, Munthe-Kaas proposed his so called Wrap Around Partitioning which is based on the reordering of the tridiagonal system of equations in such a way that the coefficient matrix appears to be a block matrix. The reordering allows to see the parallel solution of the tridiagonal system from a different perspective. The conclusions of Munthe-Kaas are that WAP(3), which corresponds to 3-CR works better than CR on vector computers. On the other hand, de Groen performed an analysis of his so called Base- p -Cyclic Reduction which is the same as R -CR. His analysis on vector computers concludes that the odd R -CR variants behave better than the even variants. The results of de Groen were obtained on Cray X-MP and NEC SX-2 vector processors. In this paper, de Groen only analyzed the problem of the strides but he did not study other problems related to R -CR like the exploitation of locality.

An even further result for tridiagonal systems on vector computers was achieved by Dodson and Levin in 1992 [DoLe92]. In their paper, they propose a 3-CR algorithm with a variation in which Cramer's rule is used to decouple two out of every three equations per step. They call this algorithm Tricyclic Reduction (TR) and they achieve a maximum exploitation of locality and a minimum execution time on one processor of the Convex C-240. In the paper, the authors claim that TR was first used in 1980 to speed up an implicit finite difference program on an IBM 3838 array processor.

Other authors have studied different aspects of CR. For instance, the early termination of CR for tridiagonal systems under conditions of strict diagonal dominance. After the first mentioning of this possibility by Hockney, Don Heller proved a formula that predicts the number of steps to be performed by his so called incomplete Block CR algorithm as a function of the diagonal dominance and the maximum error allowed to the solution [Hell74]. Stone gave bounds for the number of steps to be performed by CR when solving non-block tridiagonal systems with strict diagonal dominance [Ston75] and, Hockney and Jesshope did the same in [HoJe81]. Hegland proposed an early terminated version of the Wrap Around Partitioning of Munthe-Kaas in [Hegl91].

A different approach to CR was that described by Reuter in 1988 [Reu88]. Reuter adapted CR to exploit some locality of the computations with the vector data cache of one vector processor of the IBM 3090. His approach was similar to that by Kershaw in [Kers82] in the fact that he applied reorderings of the matrix to access it with maximum spatial locality all the time.

The use of CR on parallel computers has also been analyzed in some papers. With a similar approach to

that by Heller for the simultaneous decoupling of equations at each step [Hell74], Hockney and Jesshope proposed a variant of CR for its use on array processors [HoJe81]. This version exploits the maximum parallelism of the algorithm during all the steps of the forward phase because all the elements, and not just the even ones, are decoupled at each step during phase one. With this, the solution to all equations is computed during phase one and it is not necessary to compute the backward phase. They call this variant PARACR as opposed to the classic CR that they call SERICR. With this same idea in mind, Kapur and Brown studied the same parallel approach for block tridiagonal systems on reconfigurable array computers [KaBr81, KaBr84]. They call this approach Cyclic Elimination as opposed to the classic Cyclic Reduction in which the parallelism varies. Krechel, Plum and Stüben also compare PARACR and SERICR on one processor of the Intel iPSC2 together with other variants of CR for their use within a substructuring technique to solve multiple tridiagonal systems on parallel computers [KrPS90].

In 1990, Reale proposed the parallel use of CR on an array of T800 transputers [Real90] but he did not compare the algorithm implemented to any other competitive algorithm. Also, Johnsson analyzed the arithmetic and communication complexity of CR on different types of parallel computers that he calls ensemble architectures for tridiagonal systems [John87] and for narrow banded systems [John85, John86]. Johnsson and Ho studied the use of CR for the parallel solution of multiple tridiagonal systems of equations in [JoHo87]. Gallopoulos and Saad propose a parallel implementation of the Block Cyclic Reduction for the solution of elliptic equations in [GaSa89]. Evans and Yousif compare Cyclic Reduction and 3-Cyclic Reduction on a Balance 8000 multiprocessor with 12 processors [EvYo94]. They show that 3-CR is faster than CR for the solution of Toeplitz unsymmetric systems on that type of parallel computer but they do not compare these methods to any other competitive method. Bekakos and Evans analyze the use of CR on an MIMD computer for the periodic tridiagonal case [BeEv93]. Their study compares CR with Gaussian elimination and analyzes the percentage of time dedicated by the algorithms to communication and computation.

Other papers have dealt with the solution of special tridiagonal systems by Cyclic Reduction like that by Boisvert [Bois91]. Boisvert compares the use of CR with other specialized methods on a scalar processor for the specific case of symmetric strictly diagonal dominant tridiagonal systems of equations. The methods compared in that paper are CR, Gaussian elimination, the specialized *LU* decomposition by Malcolm and Palmer [MaPa74] and the reversed triangular factorization by Evans [Evan72, Evan80]. The reason for comparing those methods on scalar processors is the need to solve several systems of equations at the same time with the same coefficient matrix, thus the natural parallelism is trivial and it is necessary to find a fast scalar solver. CR does not compare very well to the other methods except for the periodic cases in which it is as fast as the reversed triangular factorization.

In [BoGa94], Bondelly and Gander propose direct formulae for the computation of the values of a symmetric Toeplitz tridiagonal matrix at the different steps of the CR algorithm. By this, only the right hand side has to be processed if the same coefficient matrix has to be used at different moments.

Yalamov and Evans analyze the stability of 3-CR both for diagonal dominant and strictly diagonal

dominant tridiagonal systems of equations [YaEv94].

CR has also been studied as a pentadiagonal solver. Madsen and Rodrigue proposed a variant of CR for pentadiagonal systems for the CDC STAR-100 [MaRo77]. They compared the speed and error of their version to the scalar Gaussian elimination. Also, Levit proposed a massively parallel pentadiagonal version of the Cyclic elimination in [Levi89].

Zapata, López and Argüello propose a parallel architecture for r-ary tree based algorithms that is defined by a column of a power of r processors [ZaLA94]. A clear example of r-ary tree based algorithms is the *R*-Cyclic Reduction.

Recursive Doubling

Recursive Doubling (RD) was proposed by Stone in 1973 [Ston73] and it was the first parallel algorithm for the solution of tridiagonal systems to be proposed explicitly as so. As Ortega and Voigt say, Recursive Doubling ‘... still represents one of the very few new algorithms that have resulted from considering parallel computation ...’ [OrVo85, p. 33].

Recursive Doubling is a method for the solution of first order recurrences (also called bidiagonal systems). In order to apply the algorithm, it is necessary to first decompose the matrix of the system $Ax = b$ into its L and U factors. RD is then applied to solve the recurrences arising from this decomposition. In [Ston73], the author proposes a change of variables to transform the computation of the LU decomposition into a first order recurrence and then apply RD to perform this decomposition of the original matrix. The basic idea behind RD is to express the $2i$ th element in a sequence, in terms of the i th element. Thus for $N = 2^k$, the N th component can be computed in $\log_2 N$ steps. It is important to note that RD is only stable for strictly diagonal dominant matrices [DuRo77].

Recursive Doubling was also analyzed in [Ston75] and [LaVo75], as we said. Although the results of [Ston75] led to the conclusion that Recursive Doubling has a larger operation count than CR on vector computers and a similar operation count on parallel computers, the results of [LaVo75] were more conclusive. Lambiotte and Voigt introduced the definition of consistency for the vector implementation of an algorithm. The definition of consistency states that ‘a vector implementation of an algorithm for solving a problem of size N is said to be *consistent* if the number of elementary mathematical operations required by this implementation as a function of N is the same order of magnitude as required by the usual implementation on a serial computer’ [LaVo75, p. 314]. With this definition they proved that Recursive Doubling, which has an arithmetic count of order $O(N \log_2 N)$, is not consistent and its vector implementation is slower than other consistent methods like CR, which is $O(N)$ as Gaussian elimination. The authors proposed a modification to RD to make it consistent which consisted in applying RD to systems of size with sufficiently small N . Nevertheless, this did not make RD faster than CR.

Dubois, and Rodrigue analyzed the stability of Recursive Doubling in [DuRo77]. The authors analyze the type of matrices that make RD stable and compare its execution time with Gaussian elimination on the

CDC STAR-100. Finally, Sun, Zhang and Ni utilized Recursive Doubling to solve the reduced system that arises from the Divide and Conquer algorithm that they analyze in [SuZN92].

Divide and Conquer

The origin of the Divide and Conquer algorithm to solve tridiagonal systems of equations dates back to 1978 when Sameh and Kuck proposed two parallel algorithms for the solution of such systems [SaKu78]. The intention of the authors was to propose methods based on the use of Givens rotations to make the element eliminations. The methods proposed in [SaKu78] require the use of a limited number of processors (smaller than the order of the system) and do not fail if any of the leading principal submatrices is singular at the cost of an increase in computational time.

In 1981, Wang proposed a method based on the idea behind Divide and Conquer [Wang81]. The author proposed the partitioning of the matrix into blocks of consecutive equations in the same way as Sameh and Kuck did but the diagonalization of each block was performed with the Gaussian elimination strategy. Another difference between Wang's algorithm and Sameh and Kuck's is that the final solution is found with different reorderings of the equations during stage 3. These different reorderings of the equations for the different approaches are discussed in detail in chapter 2.

In 1984, Lawrie and Sameh proposed the use of the Divide and Conquer strategy to solve general block tridiagonal systems supposing a block elimination using the Gaussian elimination strategy [LaSa84]. They analyzed the behaviour and the arithmetic complexity of the method on parallel computers with different interconnection network topologies like the shuffle exchange, the pipelined shuffle exchange and the crossbar switch.

In 1985, Meier generalized the Divide and Conquer strategy for general non-block banded systems of equations. In that paper, Meier used Wang's strategy and the Gaussian elimination basic operation to diagonalize the different partitions. Conroy also worked on a generalization of the Divide and Conquer algorithm in a similar way to that by Meier in [Conr89].

Bar-On, proposed a Divide and Conquer algorithm in which parallelism is exploited during the solution of the reduced system by applying parallelism to solve it [Bar-87]. The author proves that the operation count of his algorithm is competitive with CR on parallel computers.

In his different papers, Johnsson and Ho have compared the Divide and Conquer strategy with other methods like CR for tridiagonal systems on different types of parallel computers with different communication network topologies [John87, JoHo87]. In his papers, Johnsson proposes the use of CR to solve the reduced system formed during stage 1 of DC. For the case of banded systems, Johnson has also studied variants of the Divide and Conquer algorithm and has compared them to block Cyclic Reduction [John85, John86]. For narrow banded systems Johnsson together with Dongarra have proposed an algorithm based on the Divide and Conquer strategy and have analyzed its complexity on different architectures [DoJo87].

Some authors propose the use of Divide and Conquer on parallel computers with vector nodes. Krechel Plum and Stüben reported this in 1989 [KrPS89, KrPS90]. They performed a profound study on the use of Divide and Conquer as a parallel strategy by using different communication strategies and compared them on the Intel iPSC2 parallel computer [KrPS89]. In [KrPS90] they add a study on the use of the vector facility of each processor to the previous analysis. There, they compare PARACR and SERICR as we said before. Cox and Knisely propose the use of Divide and Conquer with some overlapping in such a way that the need for communication is reduced with the help of the redundancy of data in each processor [CoKn91]. Bondelli rewrites the Divide and Conquer strategy for general tridiagonal systems [Bond91]. He states that his algorithm can be used on computers with different levels of parallelism (i.e. parallelism and vectorization) and that it is equivalent to the algorithm by Lawrie and Sameh under some circumstances [Bond91, p.429]. Also, he proposes the early termination of the algorithm under strict diagonal dominance [Bond91, p.428]. Mehrmann worked on block tridiagonal matrices and generalized the work by Bondelli for non-block tridiagonal systems [Mehr93].

Other papers have analyzed the use of Divide and Conquer on other types of parallel computers. Hoffmann and Potma analyze it on a Meiko Computing Surface based on Transputers T800 [HoPo90]. Kumar analyzes DC on a Butterfly computer [Kuma89]. Wang and Mou analyze both the DC algorithm and PARACR on the Connection Machine [WaMo91]. Also, Sun, Zhang and Ni propose a variant of the Divide and Conquer algorithm in which the reduced system is solved with a parallel version of Recursive Doubling [SuZN92]. They compare this algorithm with other parallel algorithms like DC with the solution of the reduced system with Gaussian elimination and the early terminated version of the algorithm for strictly diagonal dominant tridiagonal systems in which the reduced systems is not solved. A proof for the accuracy of the early termination of the method proposed by Sun et al. is given in [Zhan91]. Reiter and Rodrigue proved that for the case of strict diagonal dominance, the block Divide and Conquer algorithm can be terminated early [ReRo84]. They use this result to obtain an incomplete Cholesky decomposition that they use as a preconditioner for the Conjugate Gradient iterative solver. Finally, Larriba, Jorba and Navarro propose an early termination criterion for DC and show results of its use on vector computers in [LaJN93b].

Finally, a unified architecture for Divide and Conquer for tridiagonal systems has been proposed recently by López and Zapata [Lope94, LoZa94]. With this architecture the authors unify the data flow of the algorithms for the solution of tridiagonal systems into constant geometry algorithms. This means that all communication stages are identical which allows an efficient implementation of the algorithms on parallel computers.

Bidiagonal systems solvers

The parallel solution of bidiagonal systems or first order recurrences has also been analyzed in different papers. Most of the papers compare the most popular methods for the solution of such systems.

On one hand, Van der Vorst studied the Divide and Conquer approach from different points of view. First, he analyzed two versions of the Divide and Conquer algorithm on parallel computers and two versions on vector processors in [Vand88]. In that analysis, Van der Vorst compared the parallel versions of DC on

a Sequent Balance and the Delft Parallel Processor and, the vector versions of DC on a Cray-1, a Fujitsu Facom VP200 a Cray X-MP/48 and a CDC Cyber 205. Then, in [Vand89], he presents the parallel bit of the work presented in [Vand88]. Finally, Van der Vorst together with Dekker make an analysis of the error of the Divide and Conquer algorithm and present the results on vector processors presented in [Vand88].

Axelsson and Eijkhout compare the vector version of CR, the vector version of a Product Expansion method that they propose and, the scalar recurrence [AxEi86]. The product expansion method is based on the product expansion of a basic mathematical decomposition of the coefficient matrix. The authors show the cases in which each of the three methods works better.

On the other hand, Häfner and Schönauer compared the CR algorithm with two versions of the Divide and Conquer algorithm on the vector computers IBM 3090/VF, Fujitsu VP400-EX and Cray Y-MP8/832 [HäSc90]. Also, Overill compares CR and Gaussian elimination to compute the N th term of a first order linear recurrence in [Over90].

Blelloch et al. propose the use of a technique that they call loop raking to solve linear recurrences [BICZ92]. The authors state that this technique is the dual of strip mining and can be used to vectorize recurrences that have been partitioned in the same way as for the case of Divide and Conquer.

Finally, Larriba, Navarro, Roig and Jorba perform an analysis of R -CR and DC on vector processors in [LNRJ94]. With the study, which is part of chapter 4 of this thesis, the authors optimize the use of those solvers on vector computers and show techniques to take advantage of the potential locality of some variants of the methods.

Other algorithms and methods for tridiagonal systems

Other algorithms have been designed for the solution of tridiagonal systems of equations that do not fall into the categories overviewed above.

Van der Vorst proposed a parallel method based on a special decomposition of the matrix in such a way that the systems that arise from the decomposition can be solved with parallelism equal to two [Vand87b]. He states that the decomposition is equivalent to the two sided Gaussian elimination proposed by Babuska in [Babu72] which is similar to the method proposed to Evans and Hatzopoulos in [EvHa75].

Brugnano proposes another parallel algorithm in [Brug91]. The algorithm is defined upon a sequence of products that lead to the solution of the system in parallel. The algorithm is only compared to the scalar Gaussian elimination on a Multiputer based on T800 Transputers.

Kim and Lee proposed an algorithm for the solution of tridiagonal Toeplitz systems in [KiLe90]. The algorithm is based on the use of continued fractions to parallelize Gaussian elimination. The method is not compared to any other method in the paper. Piskoulijski proved the accuracy of this algorithm in [Pisk92].

Larriba, Jorba and Navarro propose the Overlapped Partitions Method which is a parallel method for the solution of narrow banded strictly diagonal dominant systems of equations [LaJN93]. The accuracy of the method was proposed in [JoLN92]. The work presented in those papers is part of the preliminary studies

presented in this thesis.

Müller and Scheerer proposed a method for the automatic parallelization of tridiagonal systems solvers [MüSc91]. The method is based on the decomposition of the problem into a set of partitions equal to the number of processors. The Divide and Conquer algorithm is behind the description of the method.

1.4 A brief summary of applications

There are a few applications in which tridiagonal systems have to be solved that come from very diverse fields of the computational sciences. As examples of those applications we can mention the use of the ADI iterative solver for solving partial differential equations [JoSS87, Gava84, OrVo85], the computation of Natural Cubic Splines and B-Splines [ChSh92, LaNJ94], the computation of photon statistics in lasers [GuRu93] and the solution of neuron models by domain decomposition methods [Masc91]. In the following paragraphs we overview those applications and describe the type of tridiagonal systems that have to be solved in each case.

Alternating Direction Implicit

Alternating Direction Implicit is an iterative method for the solution of partial differential equations. The method has been studied for its use in parallel computers by Johnsson, Saad and Schultz in [JoSS87], by Gannon and van Rosendale in [Gava84] by Kirk and Azmy [KiAz92] and by Ortega and Voigt [OrVo85] among others. During one step of the iterative solution of a system with ADI methods it is necessary to solve a set of multiple tridiagonal systems in each of the directions of the mesh discretization. The solution of the set of tridiagonal systems in each direction of the discretization is called half a step for the two dimensional discretization and a third of a step for the three dimensional discretization.

The number of tridiagonal systems to be solved during a half or a third of a step is usually large. In this case, there is a large amount of natural parallelism and it is advisable to assign a set of systems to each processor. It is necessary to use a fast scalar solver to perform a minimum amount of total work [Bois91]. Nevertheless, depending on the data distribution among processors and the interconnection network, it can be necessary to perform some communications between two consecutive halves or one thirds of step in order to redistribute the data and be able to apply the natural parallelism referred to above. If the cost of communication is very large related to the cost of the computation, it may be advisable to solve the set of tridiagonal systems in one of the directions in parallel applying, for instance, the Divide and Conquer strategy as proposed in [JoSS87] for different types of parallel computers.

In any case, it would be advisable to compute an LU -type decomposition of the tridiagonal coefficient matrices involved in the steps of ADI. This would save some operations during the solution of the set of tridiagonal systems assigned to each processor or set of processors.

Natural Cubic Splines and B-Splines Curve Fitting

Curve fitting is used in many applications. In the context of graphic animation, curve fitting is used in the process of inbetweening images [BaBB86]. In the context of CAD, curve fitting is used to define the shape of the surfaces that form the objects being designed [RoAd90]. Also, in the context of graphic representation, curve fitting is used in the modelization of data obtained from different phenomena, i.e. medical information, etc. [BaBB86].

The process of curve fitting is performed upon a set of given points. There are different techniques to obtain a mathematical curve model of a given set of digitized points. Among these techniques we can remark Hermite, Natural Cubic Spline and B-Spline interpolation.

The solution of strictly diagonal dominant almost Toeplitz tridiagonal systems $Ax = b$ arises during the computation of Natural Cubic Splines and B-Splines. The diagonal dominance of those systems is 2. In the computation of those systems, the coefficient matrix does not change and the right hand side vector changes for different fits. The best approach to this problem is to find an LU -type decomposition ($A = LU$), store it, and compute the solution of the bidiagonal systems $Ly = b$ and $Ux = y$ every time that the fit of a new set of data is needed. This saves a considerable amount of work and, as a consequence, of computational time either if the systems have to be solved in vector mode or in parallel mode.

Chung and Chen proposed the use of an inexact LDU decomposition and further solution of the bidiagonal systems by Recursive Doubling [ChSh92]. As we said above, Recursive Doubling is inconsistent according to the definition by Lambiotte and Voigt so, the method proposed in [ChSh92] is slower than other consistent methods.

A specific LU -type decomposition has been designed for the case of Toeplitz symmetric tridiagonal systems that gives Toeplitz L and U factors [Evan80] and is described in the following section. The particular decomposition for the case of almost Toeplitz strictly diagonal dominant systems that arise in the problem described here and the solution to the bidiagonal systems that arise from it are analyzed in chapter 6 as an example.

Computation of photon statistics in lasers

The statistical behaviour of a travelling wave optical amplifier can be described through the photon probability density function differential equation that has been analyzed in [GuRu93]. The elements of the matrix equation result from the finite difference discretization of the differential equation of that probability density function, $P_{n,m}(z)$. This function describes the number of photons m in 0 or 1 mode in the propagation co-ordinate z of the emitting amplifier device as a function of the number of photons n detected at the receiving amplifier input for the same mode.

The finite difference discretization of this problem gives rise to a tridiagonal matrix that has to be used to solve a system at each step of an iterative procedure. This leads to find the probability that m photons have been emitted given that n photons have been detected at the receiving device [GuRu93].

The typical sizes of matrices are in the order of $10^4 < N < 10^5$ and the structure of the matrix is non-Toeplitz but strictly diagonal dominant. The diagonal dominance can be controlled by the space discretization parameter. The number of iterations for the iterative procedure to converge is usually around 1000 [Guit94]. In this case it is obvious that finding the classic LU decomposition is advantageous because considerable amount of work can be saved either if the bidiagonal systems have to be solved on parallel or vector computers.

Solution of neuron models by domain decomposition

The electrical behaviour of neurons can be modelled by partial differential equations that evolve in time [Hine84]. The spacial finite difference discretization of the PDE models gives rise to almost tridiagonal systems that have to be solved for different time steps in order to understand the electrical evolution of the neuron in time [Hine84]. The changing characteristics of the boundary conditions and the neurons being modelled makes the system matrix change for every new time step being modelled so the computation of an LU decomposition of the matrix is not advisable. The solution of those systems can be performed by direct or iterative solvers.

From another perspective, the solution of the problem by domain decomposition techniques shows that a set of neurons being simulated can be decomposed into a considerable amount of branching domains [Masc91]. Note that each neuron is formed by a large axon, a large amount of dendrites and a presynaptic region that can be decomposed into different cable-like domains. So, the discretization of the cable equations that model the different domains gives rise to a large number of tridiagonal systems that have to be solved at each time step. This problem is analyzed in detail in section 6 as an example of the application of the work proposed in this thesis.

The tridiagonal systems that have to be solved with this domain decomposition approach are strictly diagonal dominant because the PDEs that model the neurons are parabolic. Also, those systems are not necessarily Toeplitz and vary in size considerably from domain to domain. If the number of domains is large, the parallelism is natural and it is possible to assign a set of tridiagonal systems to a different processor in such a way that load balancing is achieved easily.

Chapter 2: A unified view of *R*-Cyclic Reduction and Divide and Conquer

*In this chapter we propose a unified view of the *R*-Cyclic Reduction and the Divide and Conquer families of parallel algorithms. For the case of strictly diagonal dominant systems of equations, we propose and prove generalized criteria for the early termination of the methods. Also, we analyze the parallelism and vector ability of the unified algorithm with the help of its parallel and vector versions.*

2.1 Introduction

R -Cyclic Reduction and Divide and Conquer have always been understood as different methods for the parallel/vector solution of bidiagonal and tridiagonal systems of equations. This is because the use of the Odd equations to eliminate variables from the Even equations in the description of CR dilutes the generality of the concept of partition used for the description of Divide and Conquer. In this chapter we unify both methods for the case of bidiagonal systems of equations. With the unified view proposed here one can conclude that the Divide and Conquer algorithm is a particular case of the R -Cyclic Reduction family of algorithms. This unified view was first mentioned in the talk [Larr93] for tridiagonal systems and as a preliminary paper in [LNRJ94] for bidiagonal systems.

From the point of view of the solution of strictly diagonal dominant systems of equations the methods can also be unified. The saving of some work to R -Cyclic Reduction and Divide and Conquer in this case has been called early termination of the methods. This early termination reduces the number of computations and also avoids those steps with a lower degree of parallelism at the cost of introducing some controllable error in the solution. With the unified algorithmic view of R -CR and DC it is possible to see that the amount of error caused by the early termination can be bound from a unified point of view. From this bound of the error, it is easy to deduce early termination criteria for both R -CR and DC. For the case of strictly diagonal dominant systems, we also analyze the amount of work saved to each method and deduce that the early termination of R -CR is not worthy because the savings are not significative. Also, we deduce that for DC those saving can be significative.

The parallel and vector aspects of the unified algorithm are also described in this chapter. Here, we pay attention to the natural parallelism of the unified algorithm and to the changes that have to be done to the unified algorithm in order to find the version for parallel and vector computers.

2.2 Unified view of R -CR and DC

Here, we use the description of DC to describe CR and then we generalize to R -CR. If we solve a bidiagonal system with DC, we can suppose that we choose a size of the partitions $R = 2$. During the forward phase a reduced system of size $\frac{N}{R} = \frac{N}{2}$ is formed. Now, the reduced system can be solved with any bidiagonal solver. For instance, we can choose to solve the reduced system with DC and $R = 2$ again. We will be able to perform a total of $S = \log_2 N$ recursive applications of DC to the corresponding reduced systems. This is equivalent to Cyclic Reduction.

In general, R can be chosen arbitrarily and the DC strategy can be applied to the corresponding reduced systems a total of $S = \log_R N$ times supposing that we keep R fixed over the different recursive applications of the DC strategy. Nevertheless, R can change from step to step, as Sweet proposed in [Sweet74]. We can

say that DC corresponds to one step of the R -CR family of algorithms. Conversely, if R is chosen to be large enough as to force $\log_R N = 1$ we will have that the R -CR algorithm reduces to the DC strategy. In the following, we assume that $S = 1$ is equivalent to DC and $S = \log_R N$ is equivalent to R -CR.

Traditionally, it has been assumed that the size of the partitions for R -CR is small and the size of the partitions for DC is large. This is because R -CR adapts very well to vector computers because it works with large vectors. On the other side, DC adapts very well to parallel computers because the problem can be partitioned into a number of subproblems equal to the number of processors.

At this point, we can modify the notation of the description of DC in order to unify it with R -CR. First, though, let us recall the three phases of DC. During phase 1 of DC we solve $D_p \{t_p, g_p\} = \{l_p, b_p\}$ for all partitions $1 < p \leq P$ except for the first one where we only solve $D_1 g_1 = b_1$. During phase 2 of DC we solve the reduced system $A^{(1)} x^{(1)} = b^{(1)}$ and during phase 3 we solve $x_p = g_p - x_{[p-1]}^{(1)} t_p$ again for all partitions $1 < p \leq P$ except for the first one.

So, we modify the notation of DC by introducing a superscript to denote the recursive step s of the unified algorithm as we did for the description of CR. So, we say that S steps are performed by the unified algorithm and s (the step) ranges from 0 to $S - 1$. This way, the amount of partitions that can be formed at each recursive step of the algorithm is $P^{(s)} = N/R^{s+1}$. The original system is $A^{(0)} x^{(0)} = b^{(0)}$ and the reduced system to be solved at step s is $A^{(s)} x^{(s)} = b^{(s)}$. The forward and backward phases change into $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$ and $x_p^{(s)} = g_p^{(s)} - x_{[p-1]}^{(s+1)} t_p^{(s)}$ respectively. $x^{(s+1)}$ is the concatenation of the $P^{(s)}$ solution elements of step $s + 1$, that is $x_{[p]}^{(s+1)} = x_{p[R]}^{(s)}$ for $1 \leq p < P^{(s)}$.

Gaussian Elimination is also embedded within the unified algorithm. If we assume that $R = N$, then $P^{(0)} = 1$ and only $D_1^{(0)} g_1^{(0)} = b_1^{(0)}$ has to be solved which is equivalent to $A^{(0)} x^{(0)} = b^{(0)}$.

R -CR and DC are embedded in the unified algorithm of Figure 2.1. This algorithm shows an iterative version of the recursive algorithm described up to here. Loop at line 1 performs the S *forward phases* of the recursive algorithm. The updates within this loop are included to keep the notation coherent. Line 2 performs the *solution of the reduced system* formed during the last *forward phase* (step $S - 1$). Finally, loop at line 3 performs the S *backward phases*. We call loop at line 1 stage 1, loop at line 2 stage 2 and, loop at line 3 stage 3. Stage 2 is performed only in case that $S < \log_R N$. In case that $S = \log_R N$ the reduced system is of size 1_by_1 and does not need to be solved explicitly. Stage 2 is usually solved with Gaussian Elimination.

As we said, this version of the unified algorithm requires that we form the new reduced system at each step s and consequently new storage space is required at each step of stage 1. The same happens with the

solution vectors $x^{(s)}$ for steps $S - 1$ down to step 0. In practice, two vectors of size N are sufficient for the solution of the algorithm. These are vector a that stores the subdiagonal of matrix A and vector b that stores the right hand side of the system. Vector b is overwritten with the final solution to the system and vector a is modified by filling it in.

```

1 do  $s = 0, S - 1$ 
   Solve  $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$  for  $1 \leq p \leq P^{(s)}$ 
   Define  $b_{[p]}^{(s+1)} = g_p^{(s)}$  for  $1 \leq p \leq P^{(s)}$ 
   Define  $a_{[p]}^{(s+1)} = t_p^{(s)}$  for  $1 \leq p \leq P^{(s)}$ 
enddo
2 Solve_reduced_system  $A^{(S)} x^{(S)} = b^{(S)}$ 
3 do  $s = S - 1, 0, -1$ 
   Perform  $x_p^{(s)} = g_p^{(s)} - x_{[p-1]}^{(s+1)} t_p^{(s)}$  for  $1 \leq p \leq P^{(s)}$ 
enddo

```

Figure 2.1. Unified algorithm for Divide and Conquer and R -Cyclic Reduction.

The total amount of arithmetic operations performed by the unified algorithm is shown in Table 2.1. First, let us remember that the solution of a bidiagonal system with Gaussian Elimination includes a product and a subtraction per equation except for the first equation. The unified algorithm performs 2 such solutions on each partition of size R at each step of the first stage. Also, an axpy type operation (a product and an addition per element) is performed on vectors of size $R - 1$ during each step of stage 3. So, $6(R - 1)P^{(0)} = 6(R - 1)N/R$ is the amount of arithmetic operations performed by step 0 of stages 1 and 3, $6(R - 1)P^{(1)} = 6(R - 1)N/R^2$ is the amount of arithmetic operations performed by step 2, etc. So, the total amount of work for R -CR ($S = \log_R N$) is $6N(R - 1)(R^{-1} + R^{-2} + \dots + R^{-S}) = 6(N - 1)$. In case that $S < \log_R N$ steps are performed, the amount of arithmetic operations is $6N(1 - R^{-S}) + 2(N/R^S - 1)$ if the reduced system is solved with Gaussian Elimination. For the particular case of DC $S = 1$ the operation count is $6N(1 - R^{-1}) + 2(N/R - 1)$. Note that assuming a large value of R for DC, this operation count is very similar to that of R -CR.

	$S = \log_R N$	$S < \log_R N$	$S = 1$
Amount of work	$6(N - 1)$	$6N(1 - R^{-S}) + 2(N/R^S - 1)$	$6N(1 - R^{-1}) + 2(N/R - 1)$

Table 2.1. Operation count for the unified algorithm.

2.3 Early termination of the methods

In this section we analyze the unified algorithm for strictly diagonal dominant systems of equations ($\delta > 1$). The unified view of R-CR and DC given in the previous section helps in understanding that the error of the early termination for strictly diagonal dominant systems can be unified, too. The formulae given in this section are proved in the following section.

During the solution of a strictly diagonal dominant system ($\delta > 1$) using R-CR ($S = \log_R N$), the diagonal dominance of the reduced system grows from step to step during stage 1. So, the diagonal dominance after step 0 is δ^R and after step 1 is δ^{R^2} . The diagonal dominance after the last step $S - 1$ is δ^{R^S} .

Because of the growth of the diagonal dominance, it is possible to see that at a certain step s of R-CR we may drop the subdiagonal of the matrix of the reduced system. Now, the coefficient matrix of the reduced system is equal to the identity matrix. With this, the method is inexact and the approximate solution has an absolute error ε bounded by

$$(2.1) \quad \varepsilon \leq \frac{\delta^{-R^S}}{(1 - \delta^{-1})} \|b^{(0)}\|_{\infty}.$$

With (2.1), it is possible to find the minimum number of steps S_{min} to be performed by R-CR in order to achieve an absolute error less than a fixed value ε

$$(2.2) \quad S_{min} \geq \left\lceil \log_R \frac{\log(\varepsilon(1 - \delta^{-1}) / \|b^{(0)}\|_{\infty})}{\log \delta^{-1}} \right\rceil.$$

If we use DC (large R and $S = 1$), the solution of the reduced system can also be avoided. In this case, the absolute error ε is bounded by

$$(2.3) \quad \varepsilon \leq \frac{\delta^{-R}}{(1 - \delta^{-1})} \|b^{(0)}\|_{\infty}$$

which is a particular case of (2.1). The explicit value of R in (2.3) gives the minimum size per partition necessary in order to solve the system with an absolute error smaller or equal than ε without solving the reduced system. We call this value of R , R_{min}

$$(2.4) \quad R_{min} = \left\lceil \frac{\log(\varepsilon(1 - \delta^{-1}) / \|b^{(0)}\|_{\infty})}{\log \delta^{-1}} \right\rceil.$$

Moreover, if the size of the partitions R is chosen to be larger than R_{min} , some more work apart from the elimination of stage 2 can be avoided. Here, it is necessary to process R equations of $D_p^{(0)} g_p^{(0)} = b_p^{(0)}$

and only the first R_{min} equations of $D_p^{(0)} t_p^{(0)} = l_p^{(0)}$ and $x_p^{(0)} = g_p^{(0)} - x_{[p-1]}^{(1)} t_p^{(0)}$.

In the case of R -CR, only R_{min} equations have to be solved during the last step $S_{min} - 1$ if R_{min} is smaller than R . Here, we have to use (2.4) with a diagonal dominance $\delta R^{S_{min}}$ to compute R_{min} for this step.

The arithmetic operations count for the early terminated unified algorithm is equivalent to $6N(1 - R^{-S_{min}})$ for $1 < S_{min} < \lceil \log_R N \rceil$. For the case $S = 1$ steps, the arithmetic operations count is equivalent to $2N(1 - R^{-1}) + 4R_{min}N/R$ due to the fact that only R_{min} equations of $D_p^{(s)} t_p^{(s)} = l_p^{(s)}$ and $x_p^{(s)} = g_p^{(s)} - x_{[p-1]}^{(s+1)} t_p^{(s)}$ are processed per partition assuming that $R = N/P^{(0)} > R_{min}$.

	$1 < S_{min} < \lceil \log_R N \rceil$	$S = 1$
Amount of work	$6N(1 - R^{-S_{min}})$	$2N(1 - R^{-1}) + 4R_{min}N/R$

Table 2.2. Operation count for the early termination of the unified algorithm.

2.3.1 Proofs for the early termination criteria

In this section we prove 2 lemmas. Lemma 2.1 determines a bound of the solution of a system with only the first element of the right hand side vector different from zero. With the help of lemma 1, lemma 2.2 proves formula (2.1) which determines the error of the unified algorithm. Also, two corollaries are enunciated. Corollary 1.1 gives the early termination criterion for R -CR shown in (2.2) and corollary 2.2 gives the early termination criterion for DC shown in (2.4).

Lemma 2.1. Let $Ax = b$ be an $R \times R$ linear system with the following characteristics:

$$i) Ax = \begin{bmatrix} 1 & & & 0 \\ a_{[2]} & 1 & & \\ & \dots & \dots & \\ 0 & & a_{[N]} & 1 \end{bmatrix} \begin{bmatrix} x_{[1]} \\ x_{[2]} \\ \dots \\ x_{[N]} \end{bmatrix} = \begin{bmatrix} b_{[1]} \\ 0 \\ \dots \\ 0 \end{bmatrix}.$$

$$ii) \min_i (1/|a_{[i]}|) = d.$$

Then, the solution vector verifies

$$|x_{[i]}| \leq |b_{[1]}| d^{i-1}, \quad i = 1, 2, \dots, R.$$

Proof. From the definition of recurrence we have that

$$\begin{aligned} x_{[1]} &= b_{[1]} \\ a_{[i]}x_{[i-1]} + x_{[i]} &= 0, \quad i = 2, 3, \dots, R \end{aligned}$$

and from the general equation we know that $x_{[i]} = -a_{[i]}x_{[i-1]} = a_{[i]}a_{[i-1]}x_{[i-2]} = \dots$. With this we have $|x_{[i]}| = |a_{[i]}a_{[i-1]} \dots a_{[1]}x_{[1]}| \leq \delta^{-i+1}|b_{[1]}|$. \square

Lemma 2.2. Let $A^{(0)}x^{(0)} = b^{(0)}$ be an $N \times N$ linear system with the following characteristics:

$$\text{i) } A^{(0)}x^{(0)} = \begin{bmatrix} 1 & & & 0 \\ a_{[2]}^{(0)} & 1 & & \\ & \dots & \dots & \\ 0 & & a_{[N]}^{(0)} & 1 \end{bmatrix} \begin{bmatrix} x_{[1]}^{(0)} \\ \dots \\ x_{[N]}^{(0)} \end{bmatrix} = \begin{bmatrix} b_{[1]}^{(0)} \\ \dots \\ b_{[N]}^{(0)} \end{bmatrix}.$$

$$\text{ii) } \min_i (1/|a_{[i]}^{(0)}|) = \delta > 1.$$

Then, for the unified algorithm described in section 2.2, it is possible to verify that the absolute error ε of any element of the solution after dropping the subdiagonal elements of the reduced system at step s is

$$\varepsilon \leq \frac{\delta^{-R^s}}{1 - \delta^{-1}} \|b^{(0)}\|_{\infty}$$

where R is the number of elements per partition.

Proof. For this proof we take the algorithm in Figure 2.1 as a reference and make use of the notation used in its description.

For the unified algorithm, a set of $1 \leq p \leq P^{(s)} = N/R^{s+1}$ systems of the type $D_p^{(s)}\{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$ is solved at each step s ($0 \leq s \leq S-1$) of stage 1. The order of these systems is R as we said before. From lemma 1, we know that the value of element R of the solution of $D_p^{(0)}t_p^{(0)} = l_p^{(0)}$ is $|t_{p[R]}^{(0)}| \leq \delta^{-R}$ because $l_p^{(0)}$ has one only element different from zero $0 < |l_{p[1]}^{(0)}| \leq \delta^{-1}$. The diagonal dominance of $D_q^{(1)}$ is larger or equal than δ^R for $1 \leq q \leq P^{(1)} = N/R^2$. This is because $D_q^{(1)}$ is formed with elements in position R of vectors $t_p^{(0)}$ for $(q-1)R+2 \leq p \leq qR$ and matrices $D_p^{(0)}$ for $(q-1)R+1 \leq p \leq qR$. Thus, the application of lemma 1 to the $S = \log_R N$ steps of R-CR, leads to a bound of the diagonal dominance of the reduced system. This diagonal dominance is larger or equal than δ^{R^S} .

We drop the elements outside the main diagonal of the reduced system $A^{(S)} x^{(S)} = b^{(S)}$ formed at step $S-1$ because they are small (smaller or equal than δ^{-R^S}). This produces an error in the solution of the system but, on the other hand, the reduced system becomes $I\tilde{x}^{(S)} = b^{(S)}$, where I is the identity matrix. With this, the error can be denoted as $e' = x^{(S)} - \tilde{x}$. Now we bound this error.

If matrix $A^{(S)}$ is decomposed into $A^{(S)} = Q + B$, where $Q = \text{diag}(A^{(S)}) = I$, then we have that $e' = (I+B)^{-1}b^{(S)} - b^{(S)} = (I-B+B^2-B^3+\dots+(-1)^{N/R^S}B^{N/R^S}-I)b^{(S)}$ which is a finite sum where N/R^S is the order of $A^{(S)}$. Thus, we have that

$$\begin{aligned} \|e'\| &\leq (\|B\|_\infty + \|B^2\|_\infty + \|B^3\|_\infty + \dots + \|B^{N/R^S}\|_\infty) \|b^{(S)}\|_\infty \leq \\ &\leq \delta^{-R^S} (1 + \delta^{-R^S} + \delta^{-2R^S} + \dots) \|b^{(S)}\|_\infty = \frac{\delta^{-R^S}}{1 - \delta^{-R^S}} \|b^{(S)}\|_\infty \end{aligned}$$

where we take the finite sum as an infinite one. This does not affect significantly the bound but simplifies the eventual handling of the formulae.

Now it is necessary to bound $\|b^{(S)}\|_\infty$. We know that $b^{(S)}$ is formed by the elements in position R of the solution of the different $D_p^{(S-1)} g_p^{(S-1)} = b_p^{(S-1)}$.

Now, let us decompose $D_p^{(S-1)}$ into $D_p^{(i)} = I + V_p^{(i)}$ with $i = S-1$. Then, $g_p^{(i)} = (I + V_p^{(i)})^{-1} b_p^{(i)} = (1 - V_p^{(i)} + (V_p^{(i)})^2 - \dots + (-1)^{R-1} (V_p^{(i)})^{R-1}) b_p^{(i)}$. Now, using that $\|V_p^{(S-1)}\|_\infty \leq \delta^{-R^{S-1}}$, we know that after step $S-1$ during stage 3:

$$\|b^{(S)}\|_\infty = \|g_p^{(S-1)}\|_\infty \leq \frac{1 - \delta^{-R^S}}{1 - \delta^{-R^{S-1}}} \|b^{(S-1)}\|_\infty$$

where we suppose that $\|b^{(S-1)}\|_\infty \geq \max_p \|b_p^{(S-1)}\|_\infty$ for $1 \leq p \leq P^{(S-1)}$.

Now, it is necessary to bound $\|b^{(S-1)}\|_\infty$. The same would happen for the rest of steps down to step 0. So, using the same procedure as that to bound $\|b^{(S)}\|_\infty$, it is not difficult to see that the bound obtained for $\|b^{(j)}\|_\infty$ in step j is

$$\|b^{(j)}\|_\infty \leq \frac{1 - \delta^{-R^j}}{1 - \delta^{-R^{j-1}}} \|b^{(j-1)}\|_\infty.$$

Then the final bound of $\|b^{(S)}\|_\infty$ is obtained with the product of the bounds for the S steps

$$\|b^{(S)}\|_\infty \leq \frac{1 - \delta^{-R^S}}{1 - \delta^{-1}} \|b^{(0)}\|_\infty$$

With this, we obtain the proposition. \square

Corollary 2.1. Given lemma 2.1, it is easy to see that it is sufficient to perform a total of S_{min} steps for the R-CR method to solve a strictly diagonal dominant bidiagonal system of equations with an absolute error less than ϵ

$$S_{min} \geq \left\lceil \log_R \frac{\log \frac{\epsilon (1 - \delta^{-1})}{\|b^{(0)}\|_\infty}}{\log \delta^{-1}} \right\rceil.$$

Corollary 2.2. Given lemma 2.1, it is easy to see that it is sufficient to have a number larger than R_{min} elements per partition for the early terminated Divide and Conquer algorithm to solve a bidiagonal system of equations with an absolute error less than ϵ without having to perform stage 2 of the algorithm

$$R_{min} \geq \left\lceil \frac{\log \frac{\epsilon (1 - \delta^{-1})}{\|b^{(0)}\|_\infty}}{\log \delta^{-1}} \right\rceil.$$

2.3.2 Analysis of the early termination of the unified algorithm

The importance on the amount of work saved to algorithms analyzed here motivates this part of the work. Both the early termination of DC and R-CR are studied and the amount of work saved in each case is quantified and compared.

Error versus diagonal dominance for the unified algorithm

A particular problem implies a diagonal dominance δ and a maximum absolute error ϵ allowed to the solution. We call this, pair (δ, ϵ) of the problem. Table 2.3 shows a few examples of pairs (δ, ϵ) and their values of S_{min} and R_{min} . Also, it is possible to understand that different pairs (δ, ϵ) determine the same number of steps S_{min} for R-CR. The same happens for the values of R_{min} for DC.

Note in Table 2.3 that for 9-CR (with $\delta = 1.1$ and $\|b^{(0)}\|_\infty = 1$) we achieve an absolute error smaller than $\epsilon = 10^{-16}$ in three steps. Nevertheless, two steps do not allow us to achieve an absolute error smaller than $\epsilon = 10^{-4}$ for that case. The same happens for other cases in Table 2.1 and this is caused by the rate of growth of the diagonal dominance at each step. During step 0, the original diagonal dominance grows from

δ to δ^R and during step 1, the diagonal dominance grows up to $\delta^{(R^2)}$. The significance of the growth of the diagonal dominance is larger for higher steps of the algorithm.

The value of R_{min} for DC varies very much depending on the pair (δ, ϵ) . Note that the original diagonal dominance grows from δ to δ^R for partitions of size R , as we said. Adding one more equation to each partition leads to a diagonal dominance δ^{R+1} . For weak diagonal dominances ($\delta \approx 1.0$) the size of the partitions has to be very large if we want the diagonal dominance to grow enough as to avoid the solution of the reduced system.

	$\delta=1.1$			$\delta=1.5$			$\delta=2.0$		
	DC	R-CR		DC	R-CR		DC	R-CR	
	R_{min}	R	S_{min}	R_{min}	R	S_{min}	R_{min}	R	S_{min}
$\epsilon = 10^{-16}$	411	2 5 9	9 4 3	93	2 5 9	7 3 3	54	2 5 9	6 3 2
$\epsilon = 10^{-7}$	194	2 5 9	8 4 3	42	2 5 9	6 3 2	24	2 5 9	5 2 2
$\epsilon = 10^{-4}$	121	2 5 9	7 3 3	25	2 5 9	5 3 2	14	2 5 9	4 2 2

Table 2.3. Values of R_{min} and S_{min} for the unified algorithm for different pairs (δ, ϵ) with $\|b^{(0)}\|_{\infty} = 1$.

As a conclusion we can say that adding one equation to each partition of DC increments the accuracy of the unified algorithm a little but adding one step to R-CR may increase its accuracy considerably. We say that the increment in accuracy is finely grained for DC and coarsely grained for R-CR.

Amount of work

As we said above, the early terminated versions of R-CR and DC avoid the solution of stage 2 and reduce the amount of work during stages 1 and 3. Here, we analyze the amount of work saved by each method.

The amount of arithmetic work performed by the early terminated R-CR is $6N(1 - R^{-S_{min}})$, as we saw. Table 2.4 shows the percentage of work performed for different values of S_{min} compared to the total amount of work for three cases of R-CR.

At the same time Table 2.3 shows that 5-CR requires from two to three steps to achieve accuracies larger than $\epsilon = 10^{-4}$ as well as 9-CR. For CR the minimum number of steps ranges from 4 to 9. It is obvious

from Table 2.4 that the early termination of R -CR does not pay off given the very little amount of savings. For 9-CR the savings add up to less than 2% and for 5-CR the savings add up to a 4% for the most optimistic cases (larger diagonal dominances and lower accuracies required). For CR the savings could be a bit larger for the most optimistic cases but the analysis in sections 4 and 5 shows that this particular case of R -CR is not the most adequate for vector and parallel computers. It is also important to say that given the very little amount of savings obtained by the early termination of the method, it does not have sense to solve R_{min} equations during the last step $S_{min} - 1$ if R_{min} is smaller than R as we suggested above.

%	$S_{min} = 1$	$S_{min} = 2$	$S_{min} = 3$	$S_{min} = 4$
CR	50	75	87.5	93.8
5-CR	80	96	99.2	99.8
9-CR	88.9	98.7	99.8	99.9

Table 2.4. Cummulative percentage of work for different number of steps S_{min} for R -CR.

For the case of DC it is possible to save some more work than for R -CR in some cases. As we saw before, the amount of work performed by DC is $2N(1 - R^{-1}) + 4R_{min}N/R$. Table 2.5 shows the percentage of work performed for different values of R_{min} and R over the total amount of work for the non-early terminated algorithm. With this Table, it is possible to see that the relative gain of the early terminated DC compared to the non-early terminated DC depends on the difference between the actual size of the partition R and the minimum size of partition R_{min} . If the difference was very large, the amount of work performed would be near a 30% of the work for the non-early terminated algorithm. Nevertheless, for moderate differences, i.e. $R_{min} = 14$ and $R = 43$, the gains are considerable (in this case a 44.9%). In general, we can say that it is possible to gain considerable amounts of work when the sizes of the partitions are a number of equations larger than R_{min} . The gain on different types of computer is quantified in sections 4 and 5.

%	$R_{min} = 14$	$R_{min} = 42$	$R_{min} = 411$
$R = 15$	97.7	not early	not early
$R = 43$	55.1	99.2	not early
$R = 412$	35.6	40.1	99.9

Table 2.5. Percentage of work compared to the non-early terminated algorithm for DC.

2.4 Parallel and Vector versions of the unified algorithm

The parallelism of the unified algorithm can be understood with the help of the unified algorithm of Figure 2.1. Now, we describe the natural parallelism of this algorithm.

Stage 1. During step s of stage 1, the $P^{(s)}$ independent partitions can be processed in parallel. The parallelism in this case is equivalent to the number of partitions and decreases at each step during this stage. Note that within each partition it is necessary to solve two recurrences to process $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$. Although the different partitions can be solved in parallel, the operations within each partition are sequential.

Stage 3. During step s of stage 3, the $P^{(s)}$ partitions can be processed independently with the same parallelism as for stage 1. Nevertheless, within each partition it is necessary to solve $x_p^{(s)} = g_p^{(s)} - x_{[p-1]}^{(s+1)} t_p^{(s)}$ which is a set of independent operations. Thus, we say that there is total parallelism at each step of stage 3.

For the parallel version of the unified algorithm, we use a message passing parallel model. Thus we suppose that it is necessary to perform a communication primitive to make two processors exchange a data item. The communication primitives are

- send (ξ, \dots, μ, π) , which sends data items ξ, \dots, μ to processor π .
- receive (ξ, \dots, μ, π) , which receives data items ξ, \dots, μ from processor π .

Those communication primitives are asynchronous. The receive primitive blocks the processor until it receives the message it is waiting for while the send primitive does not block the processor after sending the message.

Figure 2.2 shows two parallel versions of the unified algorithm. Those algorithms are the codes to be run by each processor p . The versions correspond to cases $S < \log_R N$ and $S = \log_R N$.

For both algorithms, we suppose that we have a total of $P^{(0)}$ processors which is the initial number of partitions. Also, $P^{(0)}$ is exactly equal to N/R which is a power of the size of the partition R for simplicity. During the different steps, each partition is processed by one only processor. As the amount of parallelism varies at each step the number of processors used is different at each step. So, it is necessary that the sets of R processors that share the elements of a partition of a step of stage 1 communicate to gather this partition in one of the processors. For this implementation we suppose that processors at positions iR^s for $1 \leq i \leq P^{(s)}$ are busy during each step s . So, for instance, during step 0, all processors are busy and during step 1 only processors at positions $R, 2R, 3R, \dots$ are busy, etc..

We start explaining case $S < \log_R N$ and then comment on its differences with case $S = \log_R N$.

Stage 1. The loop that performs the steps of stage 1 is divided into two parts and iterates from 0 to $S - 1$. Those processors that get into the first if construction perform some arithmetic work. This is, if the processor running the algorithm is one of the iR^s processors that have some work to do at step s (for $1 \leq i \leq P^{(s)}$),

```

1 do  $s = 0, S - 1$   $S < \log_R N$ 
  if  $p = \lfloor p/R^s \rfloor R^s$  then
    Solve  $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$ 
  endif
  if  $p = \lfloor p/R^{s+1} \rfloor R^{s+1}$  then
     $i = 1$ 
    do  $pp = R^s, R^{s+1} - R^s, R^s$ 
      receive  $(b_{p[R-i]}^{(s+1)}, a_{p[R-i]}^{(s+1)}, p - pp)$ 
       $i = i + 1$ 
    enddo
     $b_{p[R]}^{(s+1)} = g_{p[R]}^{(s)}, a_{p[R]}^{(s+1)} = t_{p[R]}^{(s)}$ 
  else if  $p = \lfloor p/R^s \rfloor R^s$  then
    send  $(g_{p[R]}^{(s)}, t_{p[R]}^{(s)}, \lceil p/R^{s+1} \rceil R^{s+1})$ 
  endif
endif
enddo
2 Solve_reduced_system  $A^{(S)} x^{(S)} = b^{(S)}$ 
3 do  $s = S - 1, 0, -1$ 
  if  $p = \lfloor p/R^{s+1} \rfloor R^{s+1}$  then
     $i = 1$ 
    do  $pp = R^s, R^{s+1} - R^s, R^s$ 
      send  $(x_{p[R-i]}^{(s+1)}, p - pp); i = i + 1$ 
    enddo
     $aux = x_{p[R-1]}^{(s+1)}$ 
  else if  $p = \lfloor p/R^s \rfloor R^s$  then
    receive  $(x_{p[R]}^{(s)}, \lceil p/R^{s+1} \rceil R^{s+1})$ 
    if  $p \neq \lceil p/R^{s+1} \rceil R^{s+1} - R^s$  then
      send  $(x_{p[R]}^{(s+1)}, p + R^s)$  endif
    if  $p \neq \lfloor p/R^{s+1} \rfloor R^{s+1} + R^s$  then
      receive  $(aux, p - R^s)$  endif
    endif
  endif
endif
enddo

```

```

1 do  $s = 0, S - 1$   $S = \log_R N$ 
  if  $p = \lfloor p/R^s \rfloor R^s$  then
    Solve  $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$ 
  endif
  if  $p = \lfloor p/R^{s+1} \rfloor R^{s+1}$  then
     $i = 1$ 
    do  $pp = R^s, R^{s+1} - R^s, R^s$ 
      receive  $(b_{p[R-i]}^{(s+1)}, a_{p[R-i]}^{(s+1)}, p - pp)$ 
       $i = i + 1$ 
    enddo
     $b_{p[R]}^{(s+1)} = g_{p[R]}^{(s)}, a_{p[R]}^{(s+1)} = t_{p[R]}^{(s)}$ 
  else if  $p = \lfloor p/R^s \rfloor R^s$  then
    send  $(g_{p[R]}^{(s)}, t_{p[R]}^{(s)}, \lceil p/R^{s+1} \rceil R^{s+1})$ 
  endif
endif
enddo
3 do  $s = S - 2, 0, -1$ 
  if  $p = \lfloor p/R^{s+1} \rfloor R^{s+1}$  then
     $i = 1$ 
    do  $pp = R^s, R^{s+1} - R^s, R^s$ 
      send  $(x_{p[R-i]}^{(s+1)}, p - pp); i = i + 1$ 
    enddo
     $aux = x_{p[R-1]}^{(s+1)}$ 
  else if  $p = \lfloor p/R^s \rfloor R^s$  then
    receive  $(x_{p[R]}^{(s)}, \lceil p/R^{s+1} \rceil R^{s+1})$ 
    if  $p \neq \lceil p/R^{s+1} \rceil R^{s+1} - R^s$  then
      send  $(x_{p[R]}^{(s+1)}, p + R^s)$  endif
    if  $p \neq \lfloor p/R^{s+1} \rfloor R^{s+1} + R^s$  then
      receive  $(aux, p - R^s)$  endif
    endif
  endif
endif
enddo

```

Figure 2.2. Parallel versions of the unified algorithm for cases $S < \log_R N$ and $S = \log_R N$.

then it performs $D_p^{(s)} \{t_p^{(s)}, g_p^{(s)}\} = \{l_p^{(s)}, b_p^{(s)}\}$. In the second if construction, all processors that work during step $s + 1$ receive the data to operate with. This is, if the processor running the algorithm is one of the iR^{s+1} processors that work during step $s + 1$ (for $1 \leq i \leq P^{(s+1)}$), then it receives the equations that still keep the recursion from the $R - 1$ processors that have one equation of the partition to be solved.

Stage 2. The reduced system is distributed among $P^{(S)} > 1$ processors that have to communicate to solve it both in case that this system is solved in a distributed manner or it is solved sequentially in one processor. The code to be run by each processor to solve the reduced system is shown in Figure 2.3 for both approaches. At the end of this step the final solutions to each equation of the reduced system have to be in the processors that originally had them.

Distributed solution of the reduced system

```

if  $p > R^S$  and  $p = \lceil p/R^S \rceil R^S$  then
    receive ( $aux, p - R^S$ )
     $x_{p[1]}^{(S)} = b_{p[1]}^{(S)} - aux \cdot a_{p[1]}^{(S)}$ 
    do  $i=1, R-1$ 
         $x_{p[i+1]}^{(S)} = b_{p[i+1]}^{(S)} - x_{p[i]}^{(S)} a_{p[i+1]}^{(S)}$ 
    enddo
endif
if  $p < P^{(0)}$  and  $p = \lfloor p/R^S \rfloor R^S$  then
    send ( $x_{p[R]}^{(S)}, p + R^S$ )
endif

```

Solution of the reduced system in one processor

```

if  $p < P^{(0)}$  and  $p = \lfloor p/R^S \rfloor R^S$  then
    send ( $g_{p[1]}^{(S)}, \dots, g_{p[R]}^{(S)}, P^{(0)}$ )
    send ( $t_{p[1]}^{(S)}, \dots, t_{p[R]}^{(S)}, P^{(0)}$ )
endif
if  $p = P^{(0)}$  then
     $i = 1$ 
    do  $pp = R^S, P^{(0)} - R^S, R^S$ 
        receive ( $a_{[i]}^{(S)}, \dots, a_{[i+R]}^{(S)}, pp$ )
        receive ( $b_{[i]}^{(S)}, \dots, b_{[i+R]}^{(S)}, pp$ )
         $i = i + R$ 
    enddo
    Solve  $A^{(S)} x^{(S)} = b^{(S)}$  sequentially
     $i = 1$ 
    do  $pp = R^S, P^{(0)} - R^S, R^S$ 
        send ( $x_{[i]}^{(S)}, \dots, x_{[i+R]}^{(S)}, pp$ )
         $i = i + R$ 
    enddo
endif
if  $p < P^{(0)}$  and  $p = \lfloor p/R^S \rfloor R^S$  then
    receive ( $x_{p[1]}^{(S)}, \dots, x_{p[R]}^{(S)}, P^{(0)}$ )
endif

```

Figure 2.3. Solution of the reduced system.

Stage 3. The loop that performs the steps of stage 3 iterates from $S - 1$ to 0. During the first if construction, those processors that have the solutions to the equations of partition iR^{s+1} (for $1 \leq i \leq P^{(s+1)}$) at step $s + 1$ send these solutions to the processors that originally had the equations of that

partition at step s (processors iR^s for $1 \leq i \leq P^{(s)}$). Note that $P^{(s)} = RP^{(s+1)}$. Also, all processors involved in step s send the solution to the equation in position R of their partition to the processor that has the nearest neighbouring partition below them at that step. Finally, the processors that work during step s find the solutions to the equations that they have assigned.

The differences between the algorithm for $S < \log_R N$ and that for $S = \log_R N$ are as follows. At the very end of stage 1 of the algorithm for $S = \log_R N$ (step $\log_R N - 1$), only processor $P^{(0)}$ has some work to do. This processor has to solve a system $R \times R$ and it solves it exactly, without the need to communicate. So, after this, there is no reduced system to be solved. Stage 2 can be avoided. Also, stage 3 starts from step $S - 2$ because there is no need to receive any result related to the solution of the reduced system.

The early termination of the unified algorithm simplifies the parallel versions shown above. In this case, it is only necessary to consider the version for $S < \log_R N$ and avoid the solution of the reduced system.

Vector version of the unified algorithm

The vector characteristics of the unified algorithm can be understood with the previous description of the parallelism of the unified algorithm. In this case, each step s of stages 1 and 3 can be vectorized. Vectorization can be reached by performing the same arithmetic operation to different data. So, during stage 1, it is necessary to perform the same operation to the same set of elements of each independent partition at the same time, we call this, vectorization across partitions. During stage 3, there is total parallelism and two vectorization strategies can be performed. First, applying vectorization across partitions as during stage 1. Second, applying vectorization within partitions. Thus the length of the vector operations is $P^{(s)}$ for stage 1 and it can be either $P^{(s)}$ or R depending on the strategy chosen for stage 3. The strategy in which vectorization is found within partitions during stage 3 is that proposed by Wang for tridiagonal systems in [Wang81] while the strategy in which vectorization is found across partitions is that proposed by Sameh and Kuck in [SaKu78].

Figure 2.4 shows two vector versions of the unified algorithm. Figure 2.4.a shows the case where stage 3 of the algorithm is vectorized across partitions and Figure 2.4.b shows the case where stage 3 of the algorithm is vectorized within partitions. Now we explain the algorithm in Figure 2.4.a and then we will explain the differences with that of Figure 2.4.b.

Stage 1. At each step of this stage, loop with index p performs the same operation on all equations in position r of the set of $P^{(s)}$ partitions of that step. Loop with index p can be vectorized because all its $P^{(s)}$ iterations are on sets of data that can be processed in parallel. The updates within this loop imply no computation and are included to keep the notation coherent as for the algorithm of Figure 2.1.

Stage 2. The reduced system at this stage is solved in case that $S < \log_R N$. When $S = \log_R N$ it is not necessary to solve it as for the parallel algorithms.

Stage 3. The vectorization of loop with index p during the steps of this stage is exactly the same as for stage 1. It is important to not that loop with index r only iterates from 1 to $R-1$ because equation at position R has been solved during the previous step. Also, loop with index p only iterates from 2 to $P^{(s)}$ because partition 1 for all steps is solved during stage 1.

<p>a)</p> <pre> 1 <u>do</u> s = 0, S - 1 <u>do</u> r=1,R <u>do</u> p=1, P^(s) (*Vectorized loop*) D_{p[r]}^(s) {t_{p[r]}^(s),g_{p[r]}^(s)} = {l_{p[r]}^(s),b_{p[r]}^(s)} <u>enddo</u> <u>enddo</u> Define b_[p]^(s+1) = g_{p[R]}^(s) for 1 ≤ p ≤ P^(s) Define a_[p]^(s+1) = t_{p[R]}^(s) for 1 ≤ p ≤ P^(s) <u>enddo</u> 2 Solve_reduced_system A^(S) x^(S) = b^(S) 3 <u>do</u> s = S - 1, 0, -1 <u>do</u> r=1,R-1 <u>do</u> p=2, P^(s) (*Vectorized loop*) x_{p[r]}^(s) = g_{p[r]}^(s) - x_[p-1]^(s+1) t_{p[r]}^(s) <u>enddo</u> <u>enddo</u> <u>enddo</u> </pre>	<p>b)</p> <pre> 1 <u>do</u> s = 0, S - 1 <u>do</u> r=1,R <u>do</u> p=1, P^(s) (*Vectorized loop*) D_{p[r]}^(s) {t_{p[r]}^(s),g_{p[r]}^(s)} = {l_{p[r]}^(s),b_{p[r]}^(s)} <u>enddo</u> <u>enddo</u> Define b_[p]^(s+1) = g_{p[R]}^(s) for 1 ≤ p ≤ P^(s) Define a_[p]^(s+1) = t_{p[R]}^(s) for 1 ≤ p ≤ P^(s) <u>enddo</u> 3 <u>do</u> s = S - 1, 0, -1 <u>do</u> p=2, P^(s) <u>do</u> r=1,R (*Vectorized loop*) x_{p[r]}^(s) = g_{p[r]}^(s) - x_[p-1]^(s+1) t_{p[r]}^(s) <u>enddo</u> <u>enddo</u> <u>enddo</u> </pre>
---	--

Figure 2.4. Vector algorithms of the unified algorithm. a) Vectorization across partitions during stage 3. b) Vectorization within partitions during stage 3.

The differences between the algorithm in Figure 2.4.b and that explained up to now are as follows. During stage 3, all partitions are traversed sequentially from partition 2. The reduced system is solved implicitly and there is no need to solve it during stage 2. Within each partition, the solution to all equations is found with the help of the solution to equation R of the previous neighbouring partition.

The early termination of the algorithm also reduces the amount of work to the vector version of the algorithm. In this case, the reduced system should not be solved. Also, during the last step of the algorithm $S_{min} - 1$, the number of iterates of loops with index r should be R for $D_{p[r]}^{(s)} g_{p[r]}^{(s)} = b_{p[r]}^{(s)}$ and R_{min} for $D_{p[r]}^{(s)} t_{p[r]}^{(s)} = l_{p[r]}^{(s)}$ and $x_{p[r]}^{(s)} = g_{p[r]}^{(s)} - x_{[p-1]}^{(s+1)} t_{p[r]}^{(s)}$. In this case, the length of the vector operations is R_{min} during the last step of stage 3 of algorithm 2.3.b.

Chapter 3: The Overlapped Partitions Method

In this chapter we propose the Overlapped Partitions Method (OPM) for the solution of strictly diagonal dominant bidiagonal systems. With OPM the original system is partitioned in such a way that some of the equations of neighbouring partitions overlap. Here, we propose and prove a formula to determine the amount of overlapping necessary as a function of the diagonal dominance of the system and the maximum error allowed to the solution. Also, we describe the parallel and vector implementations of the algorithm.

3.1 Introduction

In this chapter we propose the Overlapped Partitions Method which is an algorithm for the solution of narrow banded systems of equations with strict diagonal dominance $\delta > 1$. OPM was first proposed in [LaJN92] for the solution of tridiagonal systems. Nevertheless, OPM can be used to solve general banded systems of equations and its description and the proof of its accuracy for such systems can be found in [JoLN92].

OPM has its origins in the use of overlapping independent systems for the fast parallel search of first solution vectors for iterative methods [Larr90]. From that idea, OPM grew as a parallel direct solver for the solution of strictly diagonal dominant narrow banded systems. There is another approach which is similar in the fact that uses overlapping systems and is used to compute Incomplete LU or Incomplete Choleski factorizations for the preconditioning of Conjugate Gradient type methods [RaRo88, MaCh90, BrCM93]. The use of overlapping partitions has not been justified from a mathematical point of view in those papers. The origins of OPM and the use of the overlapping idea for the preconditioners appeared simultaneously and without any relationship.

In this chapter, we describe OPM for the solution of bidiagonal systems. We give a formula that determines the amount of overlapping necessary to OPM as a function of the diagonal dominance of the system and the maximum error allowed to the solution. With the help of the description of OPM we analyze the relation between the method and the preconditioners mentioned above. We describe the parallel and vector versions of OPM. Finally, we prove the formula for the amount of overlapping of OPM.

3.2 The Overlapped Partitions Method

For the description of the Overlapped Partitions Method we start assuming the same partitioning of matrix A and the same notation as that used for the description of DC in chapter 1. So, as a remainder, we show the partitioning of the original system $Ax = b$ and the notation in Figure 3.1. Note that we assume $\delta > 1$ and N to be divisible by an arbitrary number R for simplicity.

$$Ax = \begin{bmatrix} 1 & & & 0 \\ a_{[2]} & 1 & & \\ & \dots & \dots & \\ 0 & & a_{[N]} & 1 \end{bmatrix} x = \begin{bmatrix} D_1 & & & 0 \\ L_2 & D_2 & & \\ & \dots & \dots & \\ 0 & & L_p & D_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_p \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_p \end{bmatrix}$$

$$l_p = L_p[j, R] = \begin{bmatrix} a_{[i+1]} \\ 0 \\ \dots \\ 0 \end{bmatrix}, \forall j$$

Figure 3.1. Notation used for the description of OPM.

Solving the independent systems $D_p g_p = b_p$ for $1 \leq p \leq P$, and gathering their solutions we form $\bar{x} = (g_1, \dots, g_p, \dots, g_P)$. Vector \bar{x} is an approximation to the solution x . This approximation \bar{x} is not correct as the information contained in vectors l_p has been omitted. This is shown in Figure 3.2. So, each of the independent solutions g_p incurs in a certain amount of error (figure 3.2.b). Depending on the diagonal dominance δ of the system and the order of the partitions R , this error can be very small for the elements that are near the lower boundary of g_p and larger as the elements approach the upper boundary of g_p .

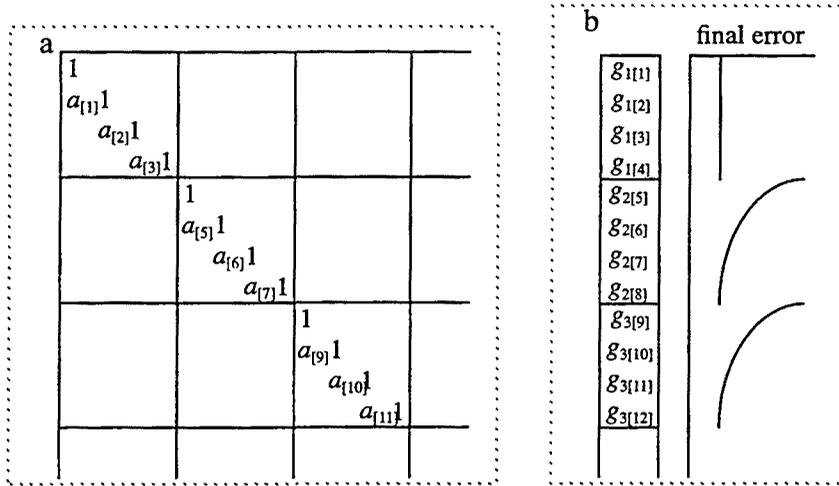


Figure 3.2. Solution and error of $D_p g_p = b_p$. a) Partitioning of $Ax = b$. b) Error of the solutions.

A possible way to reduce this error is to add m equations at the beginning of each partition except for the first partition. The new blocks $D'_p g'_p = b'_p$ overlap as shown in Figure 3.3.a for $R = 4$ and $m = 2$. So, the Overlapped Partitions Method consists in solving $D'_p g'_p = b'_p$. From the solutions to the independent partitions g'_p we take the following elements: the R elements of the first solution vector g'_1 and the R last elements of the rest of the solution vectors g'_p (for $2 \leq p \leq P$). With those elements, we form x' which is an approximation to the N elements of the solution vector x (see Figures 3.3.b and 3.3.c). An adequate choice of the overlapping m reduces the error (see Figure 3.3.c). $D'_p g'_p = b'_p$ can be solved with Gaussian Elimination although any parallel method could be used.

OPM leads to inexact solutions like the early terminated versions of R -CR and DC. The error of OPM can be controlled with the amount of overlapping m which is determined by

$$(3.1) \quad m \geq \left\lceil \frac{\log \varepsilon (1 - \delta^{-1})}{\frac{\|b\|_\infty}{\log \delta^{-1}}} \right\rceil$$

Note that this formula is exactly the same as that for DC. We prove (3.1) in the following section.

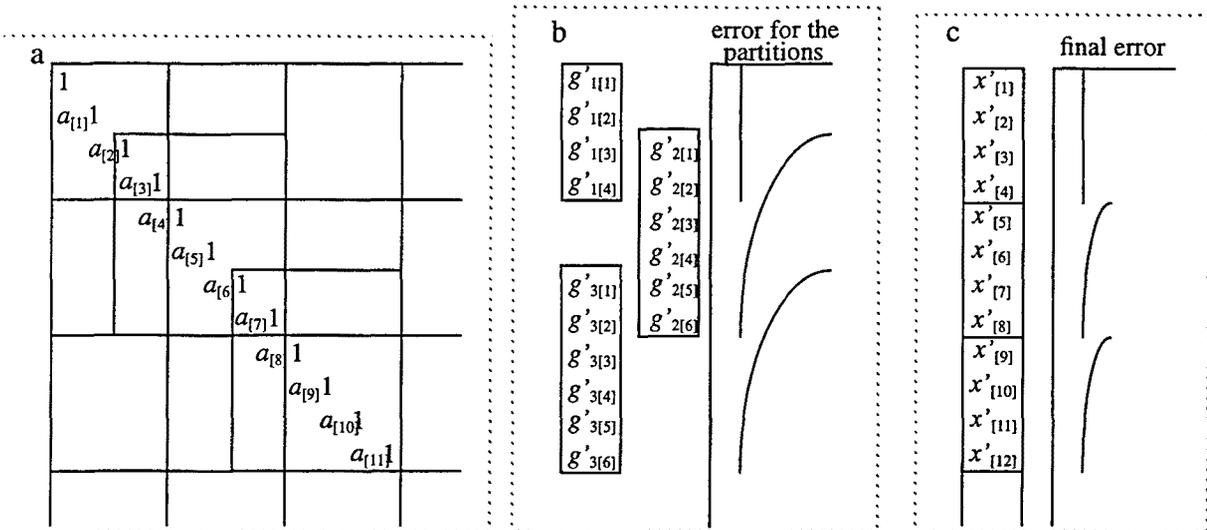


Figure 3.3. The Overlapped Partitions Method. a) Partitioning. b) Error of the partitions. c) Final solution.

The total amount of arithmetic operations performed by OPM can be described as follows. Let us recall that the solution of a bidiagonal system with Gaussian Elimination includes a product and a subtraction per equation except for the first equation. So, with the use of Gaussian Elimination to solve each partition, OPM has a total operation count of $2(N-1) + 2m(P-1)$. Remember that in case that the diagonal dominance is weak or/and the maximum error allowed is very low, m is large.

3.3 Proof for the accuracy of OPM

In this section we prove lemma 3.1 that states formula (3.1) which gives the minimum amount of overlapping to be used by OPM in order to have a certain predetermined accuracy. It is important to note that for the prove of lemma 3.1 we make use of lemma 2.1.

Lemma 3.1. Let $Ax = b$ be an $N \times N$ linear system with the following characteristics:

$$i) Ax = \begin{bmatrix} 1 & & & 0 \\ a_{[2]} & 1 & & \\ & \dots & \dots & \\ 0 & & a_{[N]} & 1 \end{bmatrix} \begin{bmatrix} x_{[1]} \\ \dots \\ x_{[N]} \end{bmatrix} = \begin{bmatrix} b_{[1]} \\ \dots \\ b_{[N]} \end{bmatrix}.$$

$$ii) \min_i (1/|a_{[i]}|) = \delta > 1.$$

Then, it is possible to verify that the necessary amount of overlapping m for OPM in order to have an absolute error less than ϵ

$$m \geq \left\lceil \frac{\log \epsilon (1 - \delta^{-1})}{\frac{\|b\|_{\infty}}{\log \delta^{-1}}} \right\rceil.$$

Proof. We take a general partition of the system $Ax = b$, $D'_p g'_p = b'_p$ as explained in section 4. The indices for this general partition range from 1 to $R + m$.

The error of the elements of g'_p with respect to the equivalent elements of the actual solution x_p is determined by $e = x_p - g'_p$. We know that e satisfies $D'_p e = c$ and $c = [c_{[1]} \ 0 \ \dots \ 0]^T = [l'_{p[1]} x_{p[0]} \ 0 \ \dots \ 0]^T$. Only the first component of l'_p is different from 0.

From lemma 2.1 we know that the value of the m th component of the error vector is bounded by $|e_m| \leq \delta^{-m+1} \|c\|_\infty$ for strictly diagonal dominant systems of equations. We also know that $\|c\|_\infty = |l'_{p[1]} x_{p[0]}| \leq \delta^{-1} |x_{[0]}|$ thus, it is necessary to bound $|x_{p[0]}|$. We bound $|x_{p[0]}|$ by finding a bound for the solution $\|x\|_\infty$ of $Ax = b$.

Since $Ax = b$ is strictly diagonal dominant we can bound $x = A^{-1}b$. We can write $A = Q + B$, where $Q = \text{diag}(A) = I$. Therefore x is determined by a finite sum $x = A^{-1}b = [I + B]^{-1}b = b - Bb + B^2b - B^3b + \dots + (-1)^{m-1} B^{m-1}b$.

Due to the structure of the matrix, it is easy to see that

$$\|x\|_\infty \leq (1 + \delta^{-1} + \delta^{-2} + \dots + \delta^{-m+1}) \|b^{(0)}\|_\infty \leq \frac{\|b\|_\infty}{1 - \delta^{-1}}$$

where we take the finite sum as an infinite one. This does not affect significantly the bound but simplifies the eventual handling of the formulae.

Thus, the value of the m th component of the error is bounded by

$$|e_m| \leq \frac{\delta^{-m}}{1 - \delta^{-1}} \|b\|_\infty$$

and the explicit value of m leads to the proposition. \square

3.4 Parallel and Vector versions of OPM

Now, we discuss the parallelism and vectorizability of OPM. Note that OPM can be summed up as the solution of $D'_p g'_p = b'_p$ for $1 \leq p \leq P$. Every single partition of OPM is independent from the rest of partitions and for this reason the method has total parallelism. Nevertheless, the high parallelism of OPM

implies that some equations are duplicated and a small amount of additional storage space is required in each processor. The algorithm of Figure 3.4 shows a parallel version of OPM in which we assume that each processor p has partition $D_p g_p = b_p$ originally and some communication is required to allow it have the overlapping that has processor $p - 1$.

```

if  $p < P$  then
    send ( $D_{p[R-m+1]}, \dots, D_{p[R]}, p+1$ )
    send ( $b_{p[R-m+1]}, \dots, b_{p[R]}, p+1$ )
endif
if  $p > 1$  then
    receive ( $D'_{p-1[1]}, \dots, D'_{p-1[m]}, p-1$ )
    receive ( $b'_{p-1[1]}, \dots, b'_{p-1[m]}, p-1$ )
endif

do  $i = m, m+R$ 
     $D'_{p[i]} = D_{p[i-m+1]}$ 
     $b'_{p[i]} = b_{p[i-m+1]}$ 
enddo
Solve  $D'_p g'_p = b'_p$ 

```

Figure 3.4. Parallel version of OPM.

Given the total parallelism of OPM, the vectorizability is trivial taking into account that the overlapping equations have to be processed in a special way as we will see in chapter 4. In this case, the length of the vectors is $P - 1$ for the operations on the m overlapping equations because the first partition has no overlapping and P for the rest of the equations. An intuitive vector version of OPM is shown in Figure 3.5. In that version, loop with index i traverses the elements of each partition sequentially and loop with index j is vectorized in such a way that the same operations are performed to the different partitions with the same vector operation.

```

do  $i = 1, m+R$ 
    do  $j = 1, P$ 
         $D'_{j[i]} g'_{j[i]} = b'_{j[i]}$ 
    enddo
enddo

```

Figure 3.5. Vector version of OPM.

3.5 Other work related to OPM

It is important to note the differences between OPM used as a direct solver and the way that the overlapped partitions are used in [RaRo88, MaCh90, BrCM93] as preconditioners and for the fast parallel search of first solution vectors for iterative methods in [Larr90]. The different proposals described here were proposed quite simultaneously but make use of independent ideas that we comment below. Also, the papers commented below do not give proves of accuracy to any of the proposed algorithms but they give experimental results.

In [RaRo88], Radicatti and Robert compare four different uses of the Incomplete LU decomposition as preconditioner for the parallel Conjugate Gradient Squared algorithm. The system is partitioned into blocks of consecutive rows and each of those strips is assigned to a different processor. Two types of decompositions are compared: 1) global Incomplete LU decompositions computed in cooperation by the set of P processors and used as local preconditioners by each processor; 2) local Incomplete LU decompositions computed by each processor independently on the set of rows assigned and used locally as preconditioners. Two variants are proposed for each of those preconditioners: a) non-overlapping; b) overlapping. Thus, for case 1.a each processor solves a system that consists of its local elements coming from the global ILU decomposition and for case 1.b each processor takes some rows of the global ILU decomposition that overlap with the two neighbouring processors. Case 2.a does not need comment and case 2.b is similar to OPM. The solution of the preconditioning phase is formed by the solutions of the non-overlapping regions and the average of the solutions to the overlapping regions. The conclusions of that paper show that 2.b is better than any of the other preconditioners. In [MaCh90], the authors use preconditioner 2.b as an example for the comparison of several CG-type methods.

Several comments should be made to the use of the overlapping preconditioners of [RaRo88, MaCh90]. First, the use of the average of the solutions in the overlapping regions is counterproductive. If we look at Figure 3.2 we can see that the solutions to a system with OPM have a larger error in the boundaries (in the case of bidiagonal and in general triangular banded systems only one boundary is affected by the error but in the case of general banded systems the two boundaries are affected by the error). Thus taking an average of the solutions in the overlapping regions causes an error larger or equal than the largest. The solution would be to take only the solution to the equations which are further to the boundary of each overlapping region in such a way that those solutions with a lower error are used.

In [BrCM93], Bru Corral and Mas propose the use of overlapped multisplitting preconditioners for the Conjugate Gradient iterative solution of symmetric M -matrices. They propose the independent solution of a set of multisplitting matrices that overlap (see [O'LVh85] for further details about multisplittings). The different neighbouring matrices have half of the value of the original matrix in the overlapping as opposed to a copy of the original value that the overlappings have for OPM. An incomplete Choleski factorization is computed for each multisplitting matrix and the solutions to the independent systems are added in order to form the solution for the preconditioning. The conclusions of this paper show that the amount of overlapping does not have a large influence on the number of iterations necessary to solve the system.

In [Larr90] an initial approximate solution for iterative solvers is found with a method similar to OPM for very narrow banded systems of equations. With the method used in [Larr90], non-overlapping partitions are formed and then matrices of the same size of the non-overlapping ones overlap between each pair of neighbouring partitions. The approximate solution is found by using the central solutions to each overlapping system. The initial approximate solution reduces the number of iterations for Conjugate Gradient and Gauss-Seidel iterations. The results showed that the larger the partitions, the lower the number of iterations necessary. The amount of overlapping was not quantified in [Larr90] by any means.

Chapter 4: Bidiagonal solvers on vector processors

In this chapter we study some aspects of the assembly language implementation of R-Cyclic Reduction, Divide and Conquer and the Overlapped Partitions Method for bidiagonal systems on vector processors. Then we assume different types of architectures and build models of the methods for those architectures. Also, we compare the methods on a vector processor of the Convex C-3480.

4.1 Introduction

During the chapter, we first describe the work performed in the field of bidiagonal solvers for vector processors. After that, we describe the five different types of vector processors that we analyze here. Then, we optimize the assembly code of the different bidiagonal solvers to save load instructions. Also, we study the vector algorithms to determine the optimum size and number of partitions for each method. We build models of R -CR, DC and OPM to understand their behaviour on the types of vector computers described and compare some of those models to real executions on the Convex C-3480 as practical example. A comparison of all the methods is made in order to determine the best method depending on the characteristics of the problem.

4.2 Previous work

The papers dealing with the solution of first order linear recurrences usually study Cyclic Reduction and the two vector versions of Divide and Conquer described in chapter 2. To our knowledge, R -Cyclic Reduction has not been compared for different values of R to Divide and Conquer for bidiagonal solvers except for [LNRJ94] where some of the results of this chapter are summarized. Also, strictly diagonal dominant solvers for vector processors have not been analyzed in the bibliography although some important applications give rise to this type of systems as we said in chapter 1. In the following paragraphs, we give an account of the work presented in previous papers.

The work performed by Van der Vorst and Dekker for the solution of first order linear recurrences on vector processors has been presented in two papers [Vand88, VaDe89]. There, the authors propose three different versions of the Divide and Conquer algorithm. Those versions are based on the use of two different compacted schemes to store the subdiagonal and the right hand side of the system which consist in using a two dimensional matrix where each column stores the elements of a partition formed for DC (scheme 1) or each row stores the elements of a partition formed for DC (scheme 2). With this, it is possible to access the matrices with stride 1 and avoid accesses to memory with strides larger than 1. So, for scheme 1 they propose two variants of DC using vectorization across and within partitions and for scheme 2 they propose the use of vectorization within partitions during stage 3 using stride 1 during stage 1. In their studies, the authors state some flop rates of the algorithms and describe the variations of the algorithm on different computers like the Cray-1, the Fujitsu Facom VP100, the Cray X-MP/48, and the Cyber 205.

Axelsson and Eijkhout compare CR, Gaussian elimination and a Product Expansion method that they propose [AxEi86]. The Product Expansion method consists in computing the solution to system $Ax = b$ by decomposing matrix A into the following factors $A = I - E$ and then computing $x = A^{-1}b = (I - E^{2^s})(I - E^{2^{s-1}}) \dots (I - E^2)(I - E)b$. Product Expansion is $O(N \log N)$ and compares negatively to CR except for a very few cases in which it is faster but the difference is negligible.

Häfner and Schönauer compare CR with two versions of DC [HäSc90]. Their analysis focusses on the

study of the optimization of DC by means of a very simple model and they observe that the optimum size of the partition is approximately \sqrt{N} . Also, for CR they propose a storage scheme in which the newly created reduced systems are stored in contiguous memory positions as in [Kers82] to avoid stride problems. They compare the methods on one processor of the IBM 3090/VF, the VP400-EX and the Cray Y-MP8/832. They conclude that given the amount of effort that has to be spent in the analysis of the different methods, it is advisable that the compilers recognize the recurrences and use built-in libraries to implement them.

Bleloch et al. propose a technique called Loop Raking that is a variation of Divide and Conquer [BICZ92]. Loop Raking can be used to solve any type of linear recurrence and can be defined as the use of strip mining plus loop interchange. With their study they show that the method allows the saving of traffic with memory. Their contribution is an insight to the problem of automatic vectorization of recurrences.

4.3 Target architecture

Vector computers usually have one or several processors and a shared memory (a few have distributed memory but this option is less frequent). In this chapter we suppose that we have one only vector processor. Each vector processor has one scalar unit and one vector unit. The scalar unit usually decodes all the instructions, processes the scalar ones and dispatches the vector instructions to the vector unit to execute them. The scalar and vector units of a vector processor can work concurrently in such a way that most of the scalar instructions are hidden by the execution of vector instructions. For simplicity, we suppose that this does not happen with the scalar arithmetic and load/store instructions involved in the algorithm but it happens for the rest of scalar instructions such as control instructions.

Vector computers have their memory organized in interleaved modules. We suppose that the number of memory modules is 2^M and the memory access time (also called latency) is 2^L clock cycles. In the case of interleaved memories, the access to vector elements that are at a regular distance S (stride S) can be degraded if S is a multiple of 2^x for x larger than $M - L$. There is no degradation when the stride is a multiple of 2^x for x smaller or equal than $M - L$, or when the stride is odd. Recent work has focussed on the proposal of memory designs that avoid such degradation of the accesses to memory [VaLA92]. These designs have not been implemented although they have proved to be effective in theory.

Vector processors usually have a set of 2^b vector registers with 2^e elements per vector register (also called Maximum Vector Length). A few processors have a small and fast reconfigurable memory that acts as a set of registers. The MVL has an important impact on the problem we analyze. This happens when the length of a real vector is larger than the MVL. Then, it is necessary to operate on slices of the vector which forces the compiler to introduce a new loop in the vector code. This technique is called strip mining. In this chapter, we suppose that we have 2^e elements per vector register for the models but we have assumed $e = 7$ (a vector length of 128) for the different comparisons we have made.

A vector processor can have several arithmetic pipelined functional units (FU). Some vector processors have one multiply and one add FUs and others have a multiple number of this set of FUs. Although division

can be implemented by means of specific FUs, it is usually implemented with an algorithm that makes use of one multiply and one add FUs in such a way that the operation can be pipelined. In this case no additions or multiplications can be performed at the same time and the speed of the vector processor is considerably slowed down. In this chapter we suppose that the processor models have one add and one multiply FU.

Load and store instructions are used to move data from memory to registers (load) and from registers to memory (store). The number of paths between vector registers and memory and the way they can be used vary from processor to processor. We suppose that load and store instructions make use of the paths between memory and the vector registers as if they were pipelined functional units that we call MEM FUs. In this chapter we study four different cases, namely, 1) the processor has one only bidirectional MEM FU, 2) two bidirectional MEM FUs, 3) one load and one store MEM FUs and, 4) two load and one store MEM FUs.

The use of the FUs of a vector processor can be chained or overlapped in such a way that a number of vector instructions larger than one and less or equal than the number of FUs is executed at the same time. The overlapping of vector instructions can be performed if there are no true dependences. The chaining of vector instructions can be performed if there are true dependences. We assume that each vector register has sufficient number of ports so that vector register access conflicts do not occur in our programs. Thus we can overlap or chain the execution of several instructions that use the same source register. Also, antidependences and output dependences do not cause structural hazards and can be chained or overlapped.

Here, we suppose that the add FU can chain off results of the multiply FU in the cases mentioned above. Also, we study different cases of arithmetic FUs being able to chain off data coming from the MEM FUs (input chaining) and MEM FUs being able to chain off the results of the arithmetic FUs (output chaining).

So, the different memory-processor cases that we analyze are: Type 1) one only path to memory that only allows output chaining. Type 2) one load path that does not allow input chaining and one store path that allows output chaining. Type 3) one load path that allows input chaining and one store path that allows output chaining. Type 4) two bidirectional paths that allow input and output chaining. Type 5) two load paths that allow input chaining and one store path that allows output chaining. This is shown in Table 4.1.

	type 1	type 2	type 3	type 4	type 5
Load	-	1	1 (ic)	-	2 (ic)
Store	-	1 (oc)	1 (oc)	-	1 (oc)
Bidirectional	1 (oc)	-	-	2 (ic, oc)	-

oc
output chaining

ic
input chaining

Table 4.1. Paths to memory and chaining ability of our vector processor models.

Modelling the execution time

A vector instruction is executed in a single stream if it cannot chain off, be chained or overlapped with other vector instructions. A group of overlapped or chained vector instructions or, an instruction executed in a

single stream is called a chime [DoGK84]. A program may have different schedulings of its machine language instructions assuming that the dependences among them are preserved. The number of chimes formed in the execution of a set of vector instructions depends on the scheduling and dependences among consecutive instructions and the architecture (number of arithmetic FUs, number of MEM FUs, chaining among them, etc). Different schedulings can give different numbers of chimes for the same instructions on the same computer. Also, the same scheduling can give different numbers of chimes on different computers.

The execution time of a vector instruction is equal to the start-up time, T_{SU} seconds, plus the number of vector elements ne times the time to process each element in vector mode which is usually the vector processor cycle time, T_V seconds. That is $T = T_{SU} + ne \cdot T_V$.

The execution time of a chime is equal to $T = Q + ne \cdot T_V$ where the constant Q depends on the number of instructions in the chime, on whether the instructions are overlapped or chained and on the start-up time of the different instructions. In order to simplify the computation of Q we suppose that it is equal to $k \cdot T_{SU}$ where k is equal to the number of vector instructions in the chime and all vector instructions have the same start-up time. So, the execution time of a chime is $T = k \cdot T_{SU} + ne \cdot T_V$ assuming that $ne \leq 2^e$.

A certain vector loop such as that shown in Figure 4.1.a is modelled by $T = T_{st} + N \cdot T_e$ if the length of the vectors is smaller or equal than MVL ($N \leq 2^e$). In our model, $T_{st} = q \cdot T_{SU}$ and $T_e = F \cdot T_V$ where q is the total number of vector instructions of the loop and F is the number of chimes of the vector code.

In the case that the vectors are larger than MVL ($N > 2^e$) the loop of Figure 4.1.a requires the use of strip mining and it is changed into the code of Figure 4.1.b. The outer loop of Figure 4.1.b is the so called strip mining loop that advances in groups of 2^e elements and the inner loop corresponds to the vector code body. Our model is transformed into $T = \lceil N/2^e \rceil \cdot T_{st} + N \cdot T_e = \lceil N/2^e \rceil \cdot q \cdot T_{SU} + N \cdot F \cdot T_V$ because $\lceil N/2^e \rceil$ start-ups are performed and we consider that the overhead introduced by loop ii negligible.

For the comparison of the execution time models below we suppose that any scalar instruction takes $T_S = 4 \cdot 10^{-7}$ seconds, the start-up of any vector instruction takes $T_{SU} = 4 \cdot 10^{-7}$ seconds and the vector processor cycle time is $T_V = 4 \cdot 10^{-8}$ seconds.

<p>(a) <u>do</u> $i=1,N$ code (i) <u>enddo</u></p>	<p>(b) <u>do</u> $ii=1,N, 2^e$ <u>do</u> $i=ii, \min(N, ii + 2^e - 1)$ code (i) <u>enddo</u> <u>enddo</u></p>
---	---

Figure 4.1. Example of vectorizable code, a) vectorizable loop, b) strip mining of the loop for $N > 2^e$.

The Convex C-3480

A processor of the Convex C-3480 has the general characteristics of the vector processors described here and, in particular, of architecture type 1. It has a total of 32 memory modules ($M = 5$) and the latency of its memory is 8 cycles ($L = 3$). It has a set of 8 vector registers ($b = 3$) and 128 elements per vector register ($e = 7$). Also, it has one add and one multiply pipelined FUs and one path to memory which allows output chaining. The vector processor cycle time of the C-3480 is $T_V = 4 \cdot 10^{-8}$ seconds and, the start-up of vector instructions and the scalar computation time depend on each specific instruction but are in the range of one order of magnitude more than T_V . The rest of features of the C-3480 do not affect the behaviour of the problem analyzed here. For further details see [Conv91].

4.4 Vector algorithms

In this section we analyze the vector versions of *R*-Cyclic Reduction, Divide and Conquer and the Overlapped Partitions Method separately. The analysis of the algorithms consists in investigating how the assembly vector code of the algorithms can be optimized in order to reduce the execution time of the algorithms. With this aim we focus on two different aspects of the interaction between the algorithm and the architecture. On one hand, we study the way to reduce the number of vector loads by maximizing the locality of the algorithms. This characteristic is independent of the computer analyzed and is studied in sections “Analysis of the assembly vector code of ...”. On the other hand, we evaluate the best schedulings of instructions for each of the five types of vector computers used in sections “Scheduling of ... for vector processors”. Then, we build models of the algorithms for the five types of architectures. Finally, in order to validate the execution times modelled we compare them to the results on the C-3480.

For the optimization of the assembly codes we use two techniques. In the following paragraphs we describe them for scalar code for simplicity. The basic recursion is defined by $x_{[i]} = b_{[i]} - a_{[i]}x_{[i-1]}$ for $2 < i \leq w$. The sequential code for the linear recursion where b is overwritten with the solution is

```
do i=2,w
  b(i)=b(i)-a(i)*b(i-1)
enddo
```

It is clear that 3 loads ($b(i)$, $a(i)$ and $b(i-1)$), 1 subtraction, 1 product and 1 store per iteration are needed to perform this loop. The assembly version of this code can be improved in two different ways:

- If $b(1)$ is loaded before the loop is entered, it is possible to keep the value of b computed in iteration i in a register and reuse it in iteration $i+1$ saving one load per iteration. We call this “variable reuse through a register”. This technique is also called “predictive commoning” [O’Br92].
- If w is small and known when designing the algorithm, the loop can be fully unrolled saving some loop control overhead. With the use of loop unrolling we also utilize registers to reuse variables and save some loads.

This type of techniques are applied in vector mode along the chapter as we will explain.

The implementations of the algorithms to be evaluated here have been coded in Convex Fortran and the appropriate vectorizing directives have been used in order to help the compiler to vectorize. Finally, the assembly code generated by the compiler has been modified in order to avoid unnecessary vector instructions and give an appropriate scheduling to those instructions if necessary.

The execution times shown in the following sections were taken in multiuser mode, which means that there were constant changes of context and many users were accessing the memory from different processors. For this reason, the timings taken for the plots were the minimum among a series of runs of the algorithms in order to reduce undesired effects and be able to analyze the aspects of interest of the problem.

4.4.1 R-Cyclic Reduction

For the case of R-CR, we only consider the case $S = \log_R N$. Also, we only consider and implement the version in which vectorization is found across partitions during the steps of stage 3. The reason for this is that, as we said in chapter 2, we are going to consider R-CR for small values of R. So, the version in which vectorization within partitions is used makes use of very short vectors during stage 3 which is not sensible.

Figure 4.2 shows stages 1 and 3 of R-CR. Note that we do not solve the reduced system because it is redundant if $S = \log_R N$. Loops with index r traverse one partition at each step s . Loops with index p are vectorized in both stages and perform the same operation on different partitions with stride R^{s+1} . The strip mining loop is performed if the vector length of loop with index p which is equal to $P^{(s)} = N/R^{s+1}$ is larger than MVL of the architecture (128 in our examples). This usually happens in the first steps of R-CR.

```

S = [logRN]
do s=0,S-1
  do r=1,R-1
    do p=Rs,N,Rs+1
      m=p+rRs-1
      b(m)=b(m)-a(m)*b(m-Rs)
      a(m)=-a(m)*a(m-Rs)
    enddo
  enddo
enddo

do s=S-1,0,-1
  do r=1,R-1
    do p=Rs+1,N,Rs+1
      m=p+rRs
      b(m)=b(m)-a(m)*b(p)
    enddo
  enddo
enddo

```

Figure 4.2. Vector version of R-Cyclic Reduction.

Given that even strides can cause memory conflicts, two families of methods can be distinguished within R-CR on vector processors: the cases with even R and the cases with odd R. Cyclic Reduction (R = 2) belongs to the first family while 3-Cyclic Reduction (R = 3) belongs to the second family. We know that the accesses to memory for R-CR have strides equal to R^{s+1} being s the step $0 \leq s \leq S - 1$ and for this

reason even values of R would give rise to potentially conflictive strides. So, in general, the family with odd strides behaves better than the family with even strides on vector computers as we will see later. For this reason we only analyze the case in which odd values of R are used.

Analysis of the assembly vector code of R -Cyclic Reduction

Figure 4.3 shows the straight vector assembly pseudocode obtained from the vectorization of loops with index p of the algorithm in Figure 4.2. There, we can see that a total of 7 vector loads, 3 vector stores, 3 vector multiplies and 3 vector add-like operations are performed.

Body of the loop of stage 1	Body of the loop of stage 3
VLOAD $b(m-R^p)$, VR0	VLOAD $b(p)$, VR0
VLOAD $a(m)$, VR1	VLOAD $a(m)$, VR1
VLOAD $b(m)$, VR2	VLOAD $b(m)$, VR2
VMUL VR0, VR1, VR3	VMUL VR0, VR1, VR3
VSUB VR2, VR3, VR4	VSUB VR2, VR3, VR4
VSTORE VR4, $b(m)$	VSTORE VR4, $b(m)$
VLOAD $a(m-R^p)$, VR4	
VMUL VR4, VR1, VR3	
VNEG VR3, VR4	
VSTORE VR4, $a(m)$	

Figure 4.3. Assembly vector pseudocode of the vector version of the body of stages 1 and 3 of R -CR.

The assembly implementation of R -CR can be optimized by reducing number of vector loads (the number of stores and arithmetic instructions cannot be reduced). One way to reduce the number of load instructions is to perform the interchange of loops with index r and p and the unrolling of loops with index r as shown in Figure 4.4.a for 3-CR. Some variables can be reused through registers with this.

Figure 4.4.b shows the vector assembly pseudocode of the body of loops in Figure 4.4.a. There, it is possible to see that the unrolled version of 3-CR performs a total of 11 vector loads, 6 vector stores, 6 vector multiplies and 6 vector add-like operations. The number of vector loads over the number of vector arithmetic operations for this unrolled version of 3-CR is $11/12$ while for the non-unrolled versions this was $7/6$. Thus, if we unroll $(R - 1)$ loops, the number of vector loads over the number of arithmetic operations is $(4R - 1) / 6(R - 1)$ which tends to $2/3$ for $R \rightarrow \infty$.

Note that the number of vector registers used in the assembly version of Figure 4.4.b is 4. In general, for any value of R , the number of registers needed is 4.

Another way to reduce the number of vector loads for R -CR could be to apply the reuse of variables through registers without unrolling as explained at the beginning of this section. In this case it would be necessary to load the first vector instances of a and b outside loop with index r reusing a and b through registers. We do not explain this technique for R -CR here although it would lead to a similar gain.

The early termination of R -CR only implies a reduction in the number of steps (loop with index s). For this reason, no changes have to be made to the assembly code explained above.

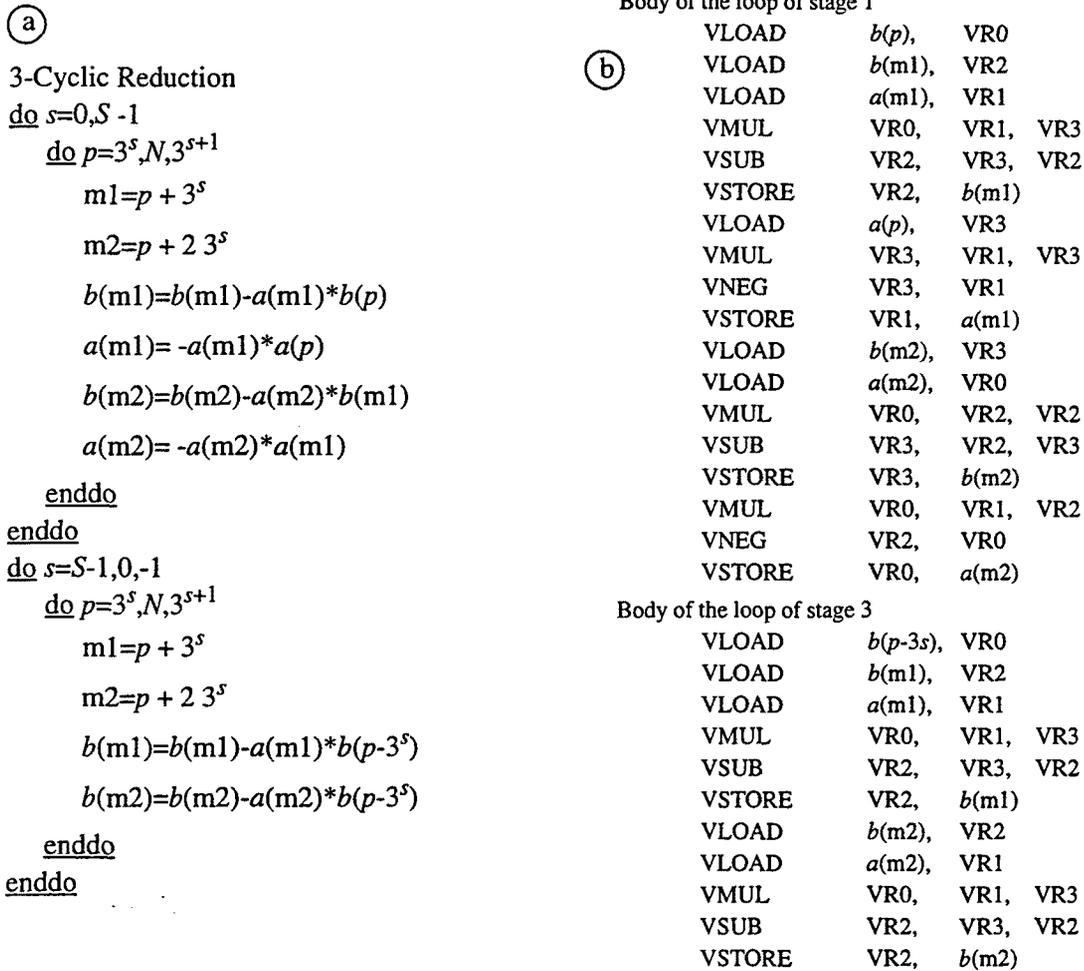


Figure 4.4. Unrolling of loop r for 3-CR. a) High Level Language version, b) Assembly version.

Scheduling of R-CR for vector processors

In this subsection we analyze R-CR for the five different types of vector processors described in section 4.3. With this aim, we count the number of chimes performed at each step of the algorithm by the bodies of the inner loops of R-CR on each type of processor given an optimum scheduling of the vector assembly instructions. The schedulings shown in Figures 4.3 and 4.4 are optimum for the non-unrolled version of R-CR and the unrolled version of 3-CR for all the vector processors analyzed.

Table 4.2 shows the number of chimes of the body of the loops of the non-unrolled and unrolled versions of R-CR. Those values have been obtained by performing a detailed study of the scheduling of those versions of R-CR.

	type 1	type 2	type 3	type 4	type 5
Chimes, non-unrolled	10	9	7	5	5
Chimes unrolled, ($R > 2$)	$F_1 = 7R - 4$	$F_2 = 5R - 1$	$F_3 = 4R$	$F_4 = 4R - 2$	$F_5 = 3R - 1$

Table 4.2. Number of chimes performed by different versions of R-CR on the architectures studied.

We illustrate the type of study performed to obtain Table 4.2 with an example showing the number of chimes formed for the unrolled version of different versions of stage 1 of R -CR on architecture type 3. This is shown in Figure 4.5 where we can deduce that the number of chimes formed unrolling $R - 1$ iterates of stage 1 is $2R + 1$ for $R > 2$. The unrolling of stage 2 would lead to $2R - 1$ chimes. The addition of the number of chimes for stage 1 and 2 leads to the value shown in Table 4.2 ($F_3 = 4R$ in our case).

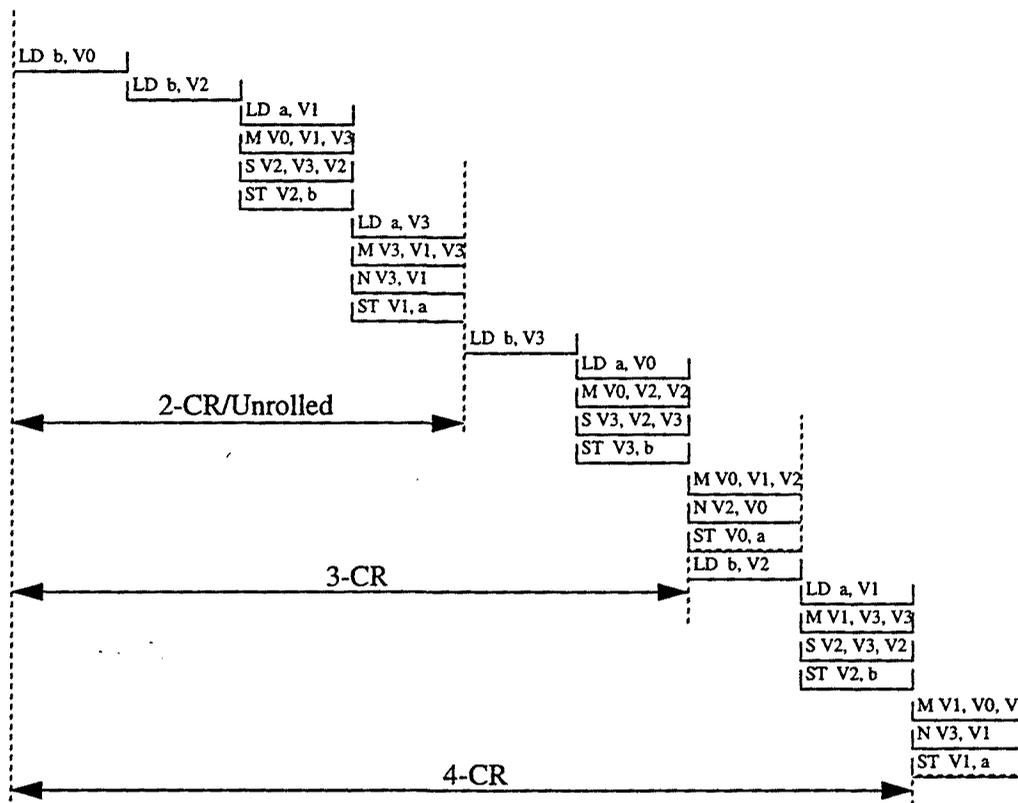


Figure 4.5. Chimes formed with the optimum scheduling of stage 1 of R -CR on vector processor type 3.

If we divide the total number of chimes of the fully unrolled versions by the number of iterates unrolled ($R - 1$) we obtain the number of chimes per iterate. This is shown in Table 4.3 together with the value of the number of chimes per iterate for $R \rightarrow \infty$.

	type 1	type 2	type 3	type 4	type 5
Chimes per iterate for $R > 2$	$3 + \frac{4R - 1}{R - 1}$	$\frac{5R - 1}{R - 1}$	$\frac{4R}{R - 1}$	$\frac{4R - 2}{R - 1}$	$\frac{3R - 1}{R - 1}$
Chimes per iterate for $R \rightarrow \infty$	7	5	4	4	3

Table 4.3. Number of chimes per iterate for R -CR on the architectures studied.

It is important to note the different gains achieved by the unrolling depending on the architecture. While for architecture type 2 the gain for the unrolled version (chimes per iterate for $R \rightarrow \infty$ in Figure 4.3) as

opposed to the non-unrolled version (chimes, non-unrolled in Figure 4.2) can be of up to a 44%, the gain for architecture type 4 can only be of up to a 20%.

A general model for R -CR

Now, we build the model of the unrolled R -CR. We first compute the execution time of one iterate of loops with index s (stage 1 + stage 2). A total of F_i chimes are performed at each step s of R -CR depending on the type of computer i (Table 4.2). Also, the number of vector instructions obtained from the vectorization of loop with index p are $3 + 13(R - 1)$ and the vector length at step s is N/R^{s+1} . With this, the execution time of a general step s of R -CR is

$$\left\lceil \frac{N}{2^e \cdot R^{s+1}} \right\rceil \cdot (3 + 13(R - 1)) \cdot T_{SU} + \frac{N}{R^{s+1}} \cdot F_i \cdot T_V$$

and the total execution time of R -CR is

$$\begin{aligned} T_{RCR} &= \sum_{s=0}^{S-1} \left[\left\lceil \frac{N}{2^e \cdot R^{s+1}} \right\rceil \cdot (3 + 13(R - 1)) \cdot T_{SU} + \frac{N}{R^{s+1}} \cdot F_i \cdot T_V \right] = \\ &= \left[\sum_{s=0}^{S-1} \left\lceil \frac{N}{2^e \cdot R^{s+1}} \right\rceil \right] \cdot (3 + 13(R - 1)) \cdot T_{SU} + N \cdot F_i \cdot T_V \end{aligned}$$

We can consider the following approximation

$$\sum_{s=0}^{S-1} \left\lceil \frac{N}{2^e \cdot R^{s+1}} \right\rceil \approx S + \left\lceil \frac{N}{(R-1)2^e} \right\rceil = \log_R N + \left\lceil \frac{N}{(R-1)2^e} \right\rceil$$

and the general model is transformed into

$$T_{RCR} \approx \left(\log_R N + \left\lceil \frac{N}{(R-1)2^e} \right\rceil \right) \cdot (3 + 13(R - 1)) \cdot T_{SU} + N \cdot F_i \cdot T_V$$

Analyzing the model for different computers

Now, we compare the performance of R -CR on the five different types of processor. On one hand, we find the optimum version of the algorithm for each computer (optimum value of R). Then, we compare the modelled execution time of R -CR to the corresponding real execution time on the C-3480. With this we validate the models.

Note that a possible way to optimize the value of R could be to find the minimum of T_{RCR} as a function of R . Nevertheless, the derivative of T_{RCR} in R leads to a function that is difficult to analyze. In order to find the optimum value of R for each type of computer we use the strategy explained below.

First, it is important to note that the advantage of reducing the number of loads only pays off for relatively small values of R . For instance for processor type 1, the number of chimes per iterate in Table 4.3

is 7.75 for 5-CR, 7.5 for 7-CR and 7.375 for 9-CR. This means that the difference between two consecutive odd R versions of R -CR is less significant for larger R . Also, if we choose large R , the vector lengths will be small, so the weight of the start ups will be significant. Thus, the optimum value of R cannot be large nor small. Note also that $N \cdot F_i \cdot T_V$ is $O(N)$ while $(\log_R N + \lfloor N / (R - 1) 2^e \rfloor) (3 + 13(R - 1)) T_{SU}$ is $O(R \log_R N)$ and for this reason the weight of $N \cdot F_i \cdot T_V$ has a considerably larger influence on the total execution time. We make a rough approximation to the optimum value of R by comparing the percentual difference of $N \cdot F_i \cdot T_V$ for two consecutive odd values of R and choosing the value of R with a difference from $R - 1$ smaller than φ . With this approach and supposing that we choose $\varphi = 0.03$ (3% difference), we obtain the optimum values of R for the different types of vector processors shown in Table 4.4.

R-CR	type 1	type 2	type 3	type 4	type 5
R	9	9	11	9	9

Table 4.4. Optimum value of R for different types of architectures for R -CR assuming $\varphi = 0.03$.

Figure 4.6 shows the execution times and speed-ups for the versions of the algorithms with the optimum values of R shown in Table 4.3 on the five different types of vector processors. For the model we have used $T_V = 4 \cdot 10^{-8}$, $T_{SU} = 4 \cdot 10^{-7}$ and $T_S = 4 \cdot 10^{-7}$ for all the architectures as we said earlier. In the global comparison of the methods in section 4.5 we will use these optimum versions. The speed-ups have been computed by dividing the execution time of each version of R -CR by the execution time of the best version of Gaussian elimination (this implies 5 scalar operations per equation). Note that the use of a processor with maximum resources (type 5) only doubles the speed-up of the processor with minimum resources (type 1).

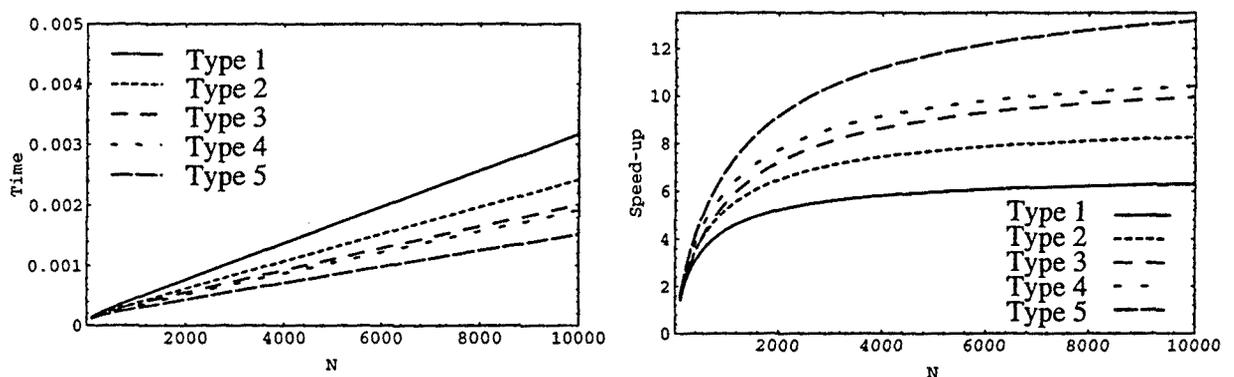


Figure 4.6. Modelled execution time and speedup of R -CR on the different types of vector processors.

Figure 4.7.a shows a comparison of the modelled time of 9-CR for architecture type 1 and the real executions on the C-3480. The model differs in less than a 6% from the real executions which allows us to say that it is accurate. Figure 4.7.b shows the execution times of CR, 3-CR, 4-CR, 5-CR, 7-CR and 9-CR on the C-3480. There, the cases with even R are considerably slower than the cases with odd R . The reason

for this, as we said, comes from the fact that strides a power of two cause memory conflicts.

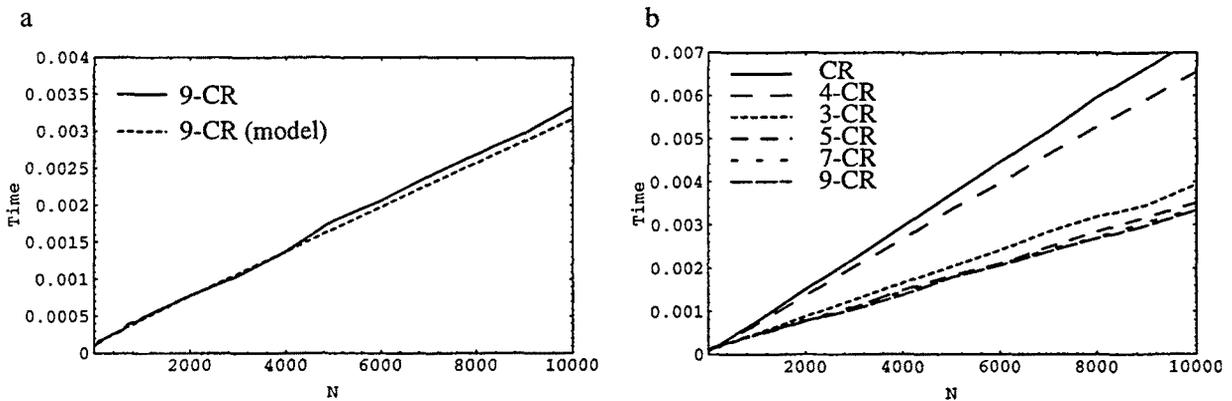


Figure 4.7. a) Comparison of the execution time of 9-CR for the model and the real execution. b) Comparison of the real executions of different versions of R-CR on the C-3480.

4.4.2 Divide and Conquer

As we said in chapter 2, there are two possible versions of the unified algorithm for vector processors. Those versions are based on the way stage 3 is vectorized and are shown in Figure 4.8.

```

a) do r=1,R-1
    do p=1,N,R (vectorized)
        m=p+r
        b(m)=b(m)-a(m)*b(m-1)
        a(m)=-a(m)*a(m-1)
    enddo
enddo
solve_reduced_system(a,b,R,N,R)
do r=1,R-1
    do p=R,N,R (vectorized)
        m=p+r
        b(m)=b(m)-a(m)*b(p)
    enddo
enddo

b) do r=1,R-1
    do p=1,N,R (vectorized)
        m=p+r
        b(m)=b(m)-a(m)*b(m-1)
        a(m)=-a(m)*a(m-1)
    enddo
enddo
do p=R,N,R
    do r=1,R (vectorized)
        m=p+r
        b(m)=b(m)-a(m)*b(p)
    enddo
enddo
    
```

Figure 4.8. Vector versions of Divide and Conquer. a) DC across partitions. b) DC within partitions.

Figure 4.8.a shows the version in which stage 3 is vectorized across partitions. This is a particular case of the algorithm for R-CR in Figure 4.2 except for the fact that routine “solve_reduced_system” is added now and only one step is performed. The parameters of routine “solve_reduced_system” are: *a* and *b*, lower diagonal and right hand side; *R*, position of the first equation of the reduced system; *N*, position of the last equation of the reduced system; *R*, distance between consecutive equations. Thus in this version of the algorithm, loops with index *r* traverse one partition during stages 1 and 3 and loops with index *p* are

vectorized. The stride of the accesses to vectors a and b is R during stages 1 and 3. In the following we call this variant of DC, DCGE because we suppose that the reduced system is solved with Gaussian elimination in scalar mode.

Figure 4.8.b shows the version in which stage 3 is vectorized within partitions. Note that stage 1 is still a particular case of the algorithm in Figure 4.2 and stages 2 and 3 are solved implicitly in the last nested loops of the algorithm. During stage 3 loops with index p and r have been interchanged. Now, loop with index p traverses the different partitions sequentially and loop with index r is vectorized. So, the stride of the accesses to memory is R during stage 1 and 1 during stage 3. We call this version DC Vector (DCV) because there is no need to perform scalar operations during its execution, except for one scalar load.

We choose R to be odd for these implementations of DC in order to avoid conflictive strides. Also, it may be necessary the use of strip mining if the vector length is larger than the vector register size.

The early terminated versions of DC are shown in Figure 4.9. Stage 1 has been partitioned into 2 loops. The first loop performs the same operations as stage 1 for the fully terminated algorithm but only on R_{min} equations. The second loop only processes the right hand side vector for the remaining $R - R_{min}$ equations. During stage 3 it is only necessary to update the first R_{min} positions of the solution vector. See chapter 2 for the computation of R_{min} .

```

a)
  do r=1, Rmin - 1
    do p=1,N,R (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(m-1)
      a(m)=-a(m)*a(m-1)
    enddo
  enddo
  do r=Rmin, R - 1
    do p=1,N,R (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(m-1)
    enddo
  enddo
  do r=1, Rmin
    do p=R, N, R (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(p)
    enddo
  enddo

b)
  do r=1, Rmin - 1
    do p=1,N,R (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(m-1)
      a(m)=-a(m)*a(m-1)
    enddo
  enddo
  do r=Rmin, R - 1
    do p=1,N,R (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(m-1)
    enddo
  enddo
  do p=R, N, R
    do r=1, Rmin (vectorized)
      m=p+r
      b(m)=b(m)-a(m)*b(p)
    enddo
  enddo

```

Figure 4.9. Vector versions of the early terminated Divide and Conquer algorithm. a) DCGE b) DCV.

Analysis of the assembly vector code of Divide and Conquer

The assembly versions of DC are very similar to the unrolled version of R -CR. The size of the partitions is usually larger for DC and, for this reason, it is not adequate to fully unroll loops with index r because the assembly code would be too large. Here we optimize DC by reusing variables through registers.

Figure 4.10 shows an optimized vector assembly pseudocode version obtained from the vectorization of the inner loops of stages 1 and 3 of DCGE and DCV. The different instances of vectors a and b of stage 1 are passed through registers VR4 and VR0 respectively across the different iterations of the loop. This is done by loading vectors a and b from position 1 to N with stride R before starting to iterate.

<p>Vector instructions performed outside the loop</p> <p>VLOAD $b(1)$, VR0</p> <p>VLOAD $a(1)$, VR4</p> <p>Body of the loop of stage 1</p> <p>VLOAD $a(m)$, VR1</p> <p>VLOAD $b(m)$, VR2</p> <p>VMUL VR0, VR1, VR3</p> <p>VSUB VR2, VR3, VR0</p> <p>VSTORE VR0, $b(m)$</p> <p>VMUL VR4, VR1, VR3</p> <p>VNEG VR3, VR4</p> <p>VSTORE VR4, $a(m)$</p>	<p>Instruction performed outside the loop</p> <p>VLOAD $b(p)$, VR0</p> <p>Body of the loop of stage 3 (across), DCGE</p> <p>VLOAD $a(m)$, VR1</p> <p>VLOAD $b(m)$, VR2</p> <p>VMUL VR0, VR1, VR3</p> <p>VSUB VR2, VR3, VR4</p> <p>VSTORE VR4, $b(m)$</p> <p>Body of the loop of stage 3 (within) DCV</p> <p>LOAD $b(p)$, R0 ; scalar</p> <p>VLOAD $a(m)$, VR1</p> <p>VLOAD $b(m)$, VR2</p> <p>VMUL R0, VR1, VR3</p> <p>VSUB VR2, VR3, VR4</p> <p>VSTORE VR4, $b(m)$</p>
---	---

Figure 4.10. Assembly vector pseudocode of the two versions of the body of stages 1 and 3 of DC.

If the length of the vectors N/R is larger than MVL strip mining has to be applied. In this case, the first instances of vectors a and b have to be loaded into more than one vector register. This is very limiting because we may require more registers than those available in the register bank to keep track of all the elements of a and b passed through registers. A possible solution to this is to perform the strip mining iterates outside the main loop as shown in Figure 4.11 for stage 1 of the algorithm. Nevertheless we do not use this approach here and we assume that the maximum number of partitions is limited to MVL.

```

do pp=1,N,R · 2e
  do r=1,R-1
    do p=pp, min (N, ii + R · 2e - 1) , R
      m=p+r
      b(m)=b(m)-a(m)*b(m-1)
      a(m)=-a(m)*a(m-1)
    enddo
  enddo
enddo

```

Figure 4.11. Strip mining loop taken outside loop with index r to apply variable reuse through registers.

Assuming that strip mining is not performed we can explain the two different versions of stage 3 of Figure 4.10. For DCGE, we load the solutions to the reduced system onto vector register VR0 before the loop starts. VR0 keeps those values along all the iterates of stage 3. For DCV it is necessary to load the solution to the last equation of partition $p - 1$ in scalar form at each iterate.

The number of vector operations performed by the two versions of DC is 2 loads, 2 stores, 2 products and 2 add-like operations for stage 1. We assume that the number of equations per partition is large, otherwise, it would be necessary to take into account the weight of the loads performed outside the body of the loop. For stage 3, the two versions perform a total of 2 loads, 1 store, 1 product and 1 add-like vector operations. For DCV, it is necessary to add one scalar load as shown in Figure 4.10.

In Figure 4.12 we show the assembly pseudocode version of stage 1 of DC for strictly diagonal dominant systems. In this piece of code, we process equations R_{min} to $R - 1$ of each partition. The rest of loops preserve the scheduling shown in Figure 4.10. With this change in the code we save 1 store, 1 product and 1 add-like operation for loops travelling from R_{min} to $R - 1$.

```

Vector instructions performed outside the loop
VLOAD      b(1),   VR0
VLOAD      a(1),   VR4
Body of the second loop of stage 1
VLOAD      a(m),   VR1
VLOAD      b(m),   VR2
VMUL       VR0,   VR1, VR3
VSUB       VR2,   VR3, VR0
VSTORE     VR0,   b(m)

```

Figure 4.12. Assembly vector pseudocode of the body of stage 1 of the early terminated DC.

Scheduling of DC for vector processors

In this subsection we analyze DC in the context of each of the five different types of vector processors described in section 4.3. With this purpose we show that it is sometimes necessary to partially unroll the inner loops of stages 1 and 3 to obtain optimum schedulings of DC for some of the types of processors used.

For the case of R -CR we unrolled all the iterates of loop with index r and for this reason we saved loads and chimes to the algorithm. For DC we do not unroll this loop completely as we said. In this case it is possible to unroll loop with index r partially to save some chimes in such a way that the loop advances in groups of U unrolled iterates as shown in Figure 4.13 for stage 1. There, we assume that $R - 1$ is divisible by U . Otherwise we would need to write an epilogue for each loop. With this, it is possible to perform a scheduling similar to that in Figure 4.4 to save some chimes. This partial unrolling only requires 4 registers.

The number of chimes for the partially unrolled versions and the non-unrolled version of DC are shown in Table 4.5. There, we show the number of chimes for U unrolled iterates of DC for stage 1, stage 3 and the addition of both. For the early termination of DC, we show the number of chimes for U unrolled iterates for equations R_{min} to $R - 1$ of stage 1 (Fe_i). Also, we show the number of chimes per iterate for each case.

Note in Table 4.5 that the partial unrolling of the different loops of DC leads to a reduction in the number of chimes which is similar to that for R-CR. This lets us foresee that both algorithms will have similar execution times.

```

do r=1,R-1,U
  do p=1,N,R
    b(p+1)=b(p+1)-a(p+1)*b(p)
    a(p+1)=-a(p+1)*a(p)
    b(p+2)=b(p+2)-a(p+2)*b(p+1)
    a(p+2)=-a(p+2)*a(p+1)
    ...
    b(p+U)=b(p+U)-a(p+U)*b(p+U-1)
    a(p+U)=-a(p+U)*a(p+U-1)
  enddo
enddo

```

Figure 4.13. Unrolling of loop r for 3-CR. a) Assembly version, b) High Level Language version.

		type 1	type 2	type 3	type 4	type 5
Chimes, non-unrolled		7	7	5	5	3
Chimes for U unrolled iterates U > 2	Stage 1	$F_{11} = 4$	$F_{21} = 3U + 1$	$F_{31} = 2U + 1$	$F_{41} = 2U + 1$	$F_{51} = 2$
	Stage 3	$F_{13} = 3$	$F_{23} = 2U + 1$	$F_{33} = 2$	$F_{43} = 2U - 1$	$F_{53} = 1$
	$F_{i1} + F_{i3}$	$F_1 = 7$	$F_2 = 5U + 2$	$F_3 = 4U + 1$	$F_4 = 4$	$F_5 = 3$
	Early Fe_i	$Fe_1 = 3$	$Fe_2 = 2U + 1$	$Fe_3 = 2$	$Fe_4 = 2U - 1$	$Fe_5 = 1$
Chimes per iterate U > 2	Stage 1	4	$(3U + 1) / U$	$(2U + 1) / U$	$(2U + 1) / U$	2
	Stage 3	3	$(2U + 1) / U$	2	$(2U - 1) / U$	1
	Stages 1+3	7	$(5U + 2) / U$	$(4U + 1) / U$	4	3
	Early	3	$(2U + 1) / U$	2	$(2U - 1) / U$	1

Table 4.5. Number of chimes performed per iterate for different versions of DC.

A general model for Divide and Conquer

Now, we build a model for the execution time of DCGE and DCV. Also, we build models for their early terminated versions. The number and size of the partitions of Divide and Conquer can be made optimum in order to perform the methods with a minimum execution time. In the following paragraphs we build the models and find the optimum number of partitions that are summarized later in Tables 4.5 and 4.6.

DCGE. The number of chimes for U unrolled iterates of the loops of stages 1 and 3 of DCGE is F_{i1} and F_{i3} respectively. We process a total of $(N/P^{(0)} - 1) / U$ sets of unrolled iterates (given that U divides

$N/P^{(0)} - 1$). The vector length is equal to the number of partitions and no strip mining is performed $P^{(0)} < 2^e$. The number of vector instructions is $8(N/P^{(0)} - 1)$ for stage 1 and $5(N/P^{(0)} - 1)$ for stage 3. So, the execution time of stages 1 and 3 is

$$13(N/P^{(0)} - 1) \cdot T_{SU} + P^{(0)} \cdot (F_{i1} + F_{i3}) \frac{N/P^{(0)} - 1}{U} \cdot T_V$$

Also, it is necessary to solve the reduced system in scalar mode. This implies 2 scalar arithmetic operations, 2 loads and 1 store if Gaussian Elimination is used. So, a total of $5(P^{(0)} - 1)$ scalar operations are performed to solve the reduced system. The complete execution time model of DCGE is

$$T_{DCGE} = [13(N/P^{(0)} - 1)] T_{SU} + [F_i(N - P^{(0)})/U] T_V + [5(P^{(0)} - 1)] T_S.$$

The optimum number of partitions obtained by minimizing this expression is

$$P^{(0)} = \sqrt{(13NT_{SU}) / (5T_S - F_i T_V / U)}.$$

DCGE_e. The early termination of DCGE implies that the reduced system is not solved. During stage 1 two loops are performed as shown in Figure 4.9. For the first loop we perform a total of F_{i1} chimes for U unrolled iterates and $8(R_{min} - 1)$ start-ups. The number of sets of unrolled iterates is $(R_{min} - 1)/U$ in this case. For the second loop of stage 1 we perform a total of F_{e_i} chimes for U unrolled iterates and $5(N/P^{(0)} - R_{min})$ start-ups. The number of sets of unrolled iterates is $(N/P^{(0)} - R_{min})/U$. During the loop of stage 3, we perform a total of F_{i3} chimes for U unrolled iterates and $5R_{min}$ start-ups. The number of sets of unrolled iterates is R_{min}/U . The vector length for all the loops is $P^{(0)}$. Thus, the execution time of DCGE_e is the addition of the execution times for the three loops mentioned

$$\begin{aligned} T_{DCGE_e} &= 8(R_{min} - 1) \cdot T_{SU} + P^{(0)} \cdot F_{i1} \frac{R_{min} - 1}{U} \cdot T_V + \\ &+ 5(N/P^{(0)} - R_{min}) \cdot T_{SU} + P^{(0)} \cdot F_{e_i} \frac{N/P^{(0)} - R_{min}}{U} \cdot T_V + \\ &+ 5R_{min} \cdot T_{SU} + P^{(0)} \cdot F_{i3} \frac{R_{min}}{U} \cdot T_V = \\ &= [5N/P^{(0)} + 8(R_{min} - 1)] T_{SU} + [F_{e_i}N + (F_i - F_{e_i})R_{min}P^{(0)} - F_{i1}P^{(0)}] T_V / U \end{aligned}$$

The optimum number of partitions for this version of DC is

$$P^{(0)} = \sqrt{5NT_{SU} / [(F_i - F_{e_i})R_{min} - F_{i1}] T_V / U}.$$

Note that it is important that the optimum size of the partitions is larger than R_{min} , otherwise DCGE

could not be terminated early.

DCV. The execution time model of stage 1 of DCV is exactly the same as that for DCGE. During stage 3, we perform a total of F_{i3} chimes for U unrolled iterates and the number of start-ups is $5P^{(0)}$. The vector length of the instructions is $N/P^{(0)}$. Also, a total of $P^{(0)}$ scalar loads have to be performed during stage 3. The execution time of DCV is

$$\begin{aligned} T_{\text{DCV}} &= 8(N/P^{(0)} - 1) \cdot T_{SU} + P^{(0)} \cdot F_{i1} \frac{N/P^{(0)} - 1}{U} \cdot T_V + \\ &\quad + 5P^{(0)} \cdot T_{SU} + N/P^{(0)} \cdot F_{i3} \frac{P^{(0)}}{U} \cdot T_V = \\ &= [8(N/P^{(0)} - 1) + 5P^{(0)}] T_{SU} + [(F_{i1} + F_{i3})N - F_{i1}P^{(0)}] T_V / U + [P^{(0)}] T_S \end{aligned}$$

The optimum number of partitions for this expression is

$$P^{(0)} = \sqrt{(8NT_{SU}) / (T_S + 5T_{SU} - F_{i1}T_V/U)}.$$

DCV_e. The execution time model of the first stage of the early termination of DCV is like that for DCGE_e. During stage 3, we perform a total of F_{i3} chimes for U unrolled iterates and the number of start-ups is $5P^{(0)}$. The vector length of the instructions is R_{min} . Also, a total of $P^{(0)}$ scalar loads have to be performed during stage 3. The execution time of DCV is

$$\begin{aligned} T_{\text{DCV}_e} &= 8(R_{min} - 1) \cdot T_{SU} + P^{(0)} \cdot F_{i1} \frac{R_{min} - 1}{U} \cdot T_V + \\ &\quad + 5(N/P^{(0)} - R_{min}) \cdot T_{SU} + P^{(0)} \cdot F_{e_i} \frac{N/P^{(0)} - R_{min}}{U} \cdot T_V + \\ &\quad + 5P^{(0)} \cdot T_{SU} + R_{min} \cdot F_{i3} \frac{P^{(0)}}{U} \cdot T_V = \\ &= [5N/P^{(0)} + 3(R_{min} - 1) + 5P^{(0)}] T_{SU} + \\ &\quad + [F_{e_i}N + (F_i - F_{e_i})R_{min}P^{(0)} - F_{i1}P^{(0)}] T_V + [P^{(0)}] T_S \end{aligned}$$

The optimum number of partitions is

$$P^{(0)} = \sqrt{5NT_{SU} / [T_S + 5T_{SU} + ((F_i - F_{e_i})R_{min} - F_{i1})/U] T_V}.$$

As we said, Tables 4.5 and 4.6 summarize the execution time models and optimum number of partitions for the different versions of DC.

	T_i
DCGE	$[13(N/P^{(0)} - 1)]T_{SU} + [F_i(N - P^{(0)})/U]T_V + [5(P^{(0)} - 1)]T_S$
DCGEe	$[5N/P^{(0)} + 8(R_{min} - 1)]T_{SU} + [F_{e_i}N + (F_i - F_{e_i})R_{min}P^{(0)} - F_{i1}P^{(0)}]T_V/U$
DCV	$[8(N/P^{(0)} - 1) + 5P^{(0)}]T_{SU} + [(F_{i1} + F_{i3})N - F_{i1}P^{(0)})/U]T_V + [P^{(0)}]T_S$
DCVe	$[5N/P^{(0)} + 3(R_{min} - 1) + 5P^{(0)}]T_{SU} + [F_{e_i}N + (F_i - F_{e_i})R_{min}P^{(0)} - F_{i1}P^{(0)}]T_V + [P^{(0)}]T_S$

Table 4.6. Models of the different versions of DC.

	$P^{(0)}$
DCGE	$\sqrt{(13NT_{SU}) / (5T_S - F_i T_V / U)}$
DCGEe	$\sqrt{5NT_{SU} / [(F_i - F_{e_i})R_{min} - F_{i1}] T_V / U}$
DCV	$\sqrt{(8NT_{SU}) / (T_S + 5T_{SU} - F_{i1} T_V / U)}$
DCVe	$\sqrt{5NT_{SU} / [T_S + 5T_{SU} + ((F_i - F_{e_i})R_{min} - F_{i1}) / U] T_V}$

Table 4.7. Optimum sizes of the partitions for the different versions of DC.

Finding an optimum value of the partial unrolling U

In order to find the optimum value of U for each type of computer we have compared the execution times given by the models shown in Tables 4.6 and 4.7 for different values of U . We choose the smallest U that gives an execution time smaller than $\varphi = 0.03$ (3% difference) compared to the execution time given by $U - 1$. The optimum values of U are shown in Table 4.8. for the different architectures studied.

DC	type 1	type 2	type 3	type 4	type 5
U for T_{DCGE}	1	4	2	2	1
U for T_{DCGEe}	1	5	4	3	1
U for T_{DCV}	1	4	2	2	1
U for T_{DCVe}	1	5	4	3	1

Table 4.8. Optimum value of U for different types of architectures and versions of DC.

Remarks on finding the optimum number of partitions on a real computer

In a practical implementation of DC, it is necessary to compute the optimum value of $P^{(0)}$ to minimize the execution time of the algorithm, as we have shown above. Nevertheless, the vendor does not usually provide the values of T_{SU} and T_V and also they may vary considerably depending on the instruction. In this case, it is possible to drop all the constants of the model of time and find their approximate values by fitting the execution time model to a set of experimental measures by least squares.

Let us explain this with an example. Suppose that we have the model for DCGE

$$T_{DCGE} = [13 (N/P^{(0)} - 1)] T_{SU} + [F_i (N - P^{(0)}) / U] T_V + [5 (P^{(0)} - 1)] T_S.$$

If we drop the constants and substitute them by variables we have

$$\bar{T}_{DCGE} = K_1 N + K_2 P^{(0)} + K_3 N/P^{(0)} + K_4.$$

With this, the optimum number of partitions is

$$P^{(0)} = \sqrt{K_3 N / K_2}.$$

Note that we assume that F_i and U are constants for a specific implementation.

So, K_1, K_2, K_3 and K_4 can be determined by executing the algorithm for systems with different values of N and $P^{(0)}$ and then fitting the model to those experimental times by, for instance, least squares.

We have proceeded in this way to obtain the optimum number of partitions in the practical implementations of DC on the Convex C-3480. These fitted models are not used for anything else but computing the optimum $P^{(0)}$.

The execution time models that we have fitted for DCV and DCGE on the C-3480 are

$$T_{DCGE} = K_1 N + K_2 P^{(0)} + K_3 \frac{N}{P^{(0)}} + K_4 \quad T_{DCV} = K_5 N + K_6 P^{(0)} + K_7 \frac{N}{P^{(0)}} + K_8$$

and the corresponding values of the constants after the fits are

$$\begin{aligned} \text{DCGE} \rightarrow K_1 &= 2.9 \cdot 10^{-7} & K_2 &= 1.5 \cdot 10^{-6} & K_3 &= 7.8 \cdot 10^{-6} & K_4 &= -5.5 \cdot 10^{-5} \\ \text{DCV} \rightarrow K_5 &= 3.0 \cdot 10^{-7} & K_6 &= 2.8 \cdot 10^{-6} & K_7 &= 4.0 \cdot 10^{-6} & K_8 &= -2.0 \cdot 10^{-5} \end{aligned}$$

The mean quadratic error of these fits is less 14% for DCGE and less than 5% for DCV. With this, the optimum values of $P^{(0)}$ are

$$\text{DCGE} \rightarrow P^{(0)} \cong 2.3 \sqrt{N} \quad \text{DCV} \rightarrow P^{(0)} \cong 1.2 \sqrt{N}.$$

The theoretical optimum values of $P^{(0)}$ are $\text{DCGE} \rightarrow P^{(0)} \cong 1.74 \sqrt{N}$ and $\text{DCV} \rightarrow P^{(0)} \cong 1.2 \sqrt{N}$ using

$T_V = 4 \cdot 10^{-8}$, $T_{SU} = 4 \cdot 10^{-7}$ and $T_S = 4 \cdot 10^{-7}$ as we said earlier. It is important to note the significative difference between the practical and theoretical values for DCGE. This is caused by the fact that different values of $P^{(0)}$ that are relatively far from the optimum give similar execution times. This gives rise to the inaccuracy of the fit for DCGE while it does not happen for DCV because the optimum values of $P^{(0)}$ give considerably different execution times than values near its optimum.

Now we find fits of the execution time models for the early terminated versions of DC

$$T_{DCGEe} = K_1 N + K_2 P^{(0)} + K_3 \frac{N}{P^{(0)}} + K_4 R_{min} + K_5 R_{min} P^{(0)} + K_6$$

$$T_{DCVe} = K_7 N + K_8 P^{(0)} + K_9 \frac{N}{P^{(0)}} + K_{10} R_{min} + K_{11} R_{min} P^{(0)} + K_{12}$$

The corresponding values of the constants after the fits for the C-3480 are

$$DCGEe \rightarrow K_1 = 1.14 \cdot 10^{-7} \quad K_2 = 6.8 \cdot 10^{-7} \quad K_3 = 2.7 \cdot 10^{-6} \quad K_4 = 2.2 \cdot 10^{-6}$$

$$K_5 = 2.0 \cdot 10^{-7} \quad K_6 = -4.3 \cdot 10^{-5}$$

$$DCVe \rightarrow K_7 = 1.16 \cdot 10^{-7} \quad K_8 = 3.53 \cdot 10^{-6} \quad K_9 = 2.6 \cdot 10^{-6} \quad K_{10} = -5.1 \cdot 10^{-5}$$

$$K_{11} = 2.0 \cdot 10^{-7} \quad K_{12} = -8.1 \cdot 10^{-5}$$

The mean quadratic error of these fits of the models is less 10% for both DCGEe and DCVe. With this, the optimum number of partitions for the algorithms on the C-3480 are

$$DCGEe \rightarrow P^{(0)} \cong \sqrt{\frac{101.1N}{25.8 + 7.6R_{min}}} \quad DCVe \rightarrow P^{(0)} \cong \sqrt{\frac{99.2N}{133.6 + 7.7R_{min}}}$$

and the theoretical optimum numbers of partitions are

$$DCGEe \rightarrow P^{(0)} \cong \sqrt{\frac{100N}{6 + 8R_{min}}} \quad DCVe \rightarrow P^{(0)} \cong \sqrt{\frac{100N}{114 + 8R_{min}}}$$

which do not differ considerably from the values for the C-3480 except for the case of one constant for DCGEe. In this case, the theoretical optimum gives a considerably smaller value than the fit. The reason for this difference is the same as that explained for the non-early terminated versions.

Analyzing the model for different computers

Now we are going to compare the performance of the algorithms on the five types of processor that we study. First we show that DCV (non-early terminated version) is potentially faster than DCGE and then we compare the execution times of DCV on the five architectures that we study. Finally, we compare the early terminated versions of the algorithms.

Figure 4.14 will help us to determine the relative speed between DCGE and DCV for the five types of processors studied. The plot shows the difference between the execution times modelled for DCGE and

DCV determined by $100(T_{DCGE} - T_{DCV})/T_{DCV}$. The size of the matrices ranges from $N = 100$ to $N = 10000$. There, it is possible to see that the execution times are slightly better for DCV than for DCGE in most of the cases. In general, DCV is more independent of the relative speed between the scalar and vector processors because the reduced system is not solved explicitly with Gaussian elimination. For this reason we choose DCV for further comparisons as it is the potentially fastest version of DC.

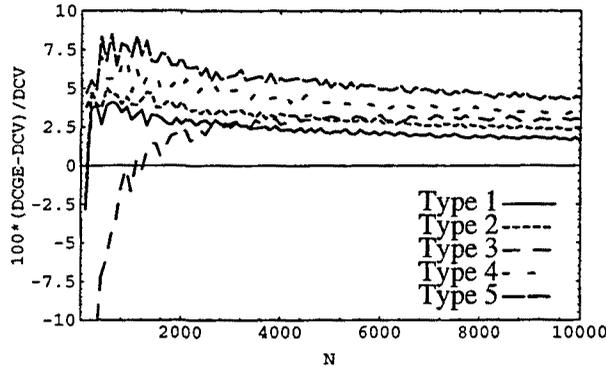


Figure 4.14. Relative difference between the modelled execution time of DCGE and DCV.

The comparison of the modelled execution times of DCV for the five types of computer studied here are shown in Figure 4.15. We assume that those execution time models are validated because the difference between the execution times modelled for architecture type 1 and the real executions of DCV and DCGE on the C-3480 is less than 5%.

For the plots of Figure 4.15 and in the following we have forced the models to use an optimum number of partitions smaller than the vector length ($P^{(0)} \leq 2^e$) because, as we said, we avoid strip mining. For this reason we choose $P^{(0)} = 2^e$ if the optimum value of $P^{(0)}$ given by the formula is larger than 2^e .

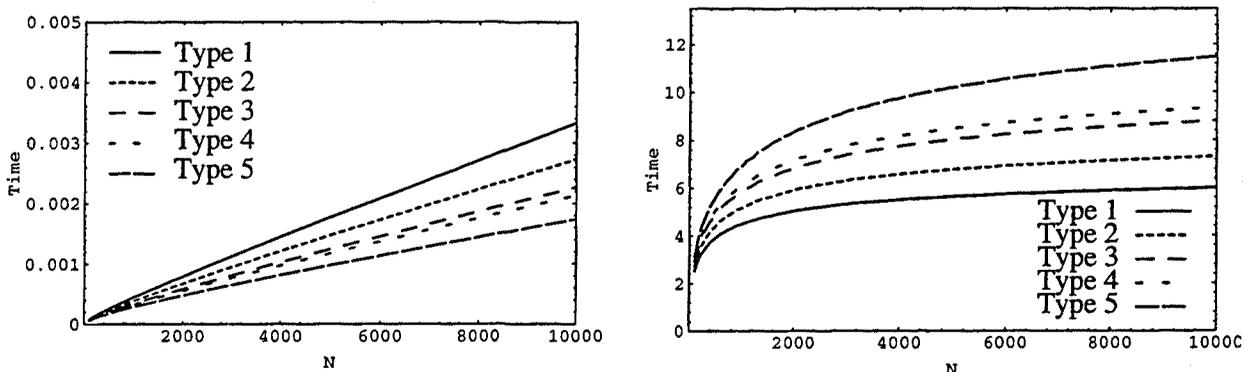


Figure 4.15. Modelled execution time and speedup of DCV on the five types of vector processor.

Now, we are going to analyze the early terminated versions of DC. Here, though, we do not compare the five different models of computers and restrict to processor type 1 and its comparison to the real executions

on one processor of the C-3480. Our aim now is the validation of the model for this case. In section 4.5 we make a global comparison for the other types of processors.

For the early terminated versions of DC we also suppose that if the optimum number of partitions is larger than the vector length we choose $P^{(0)} = 2^e$.

Figure 4.16 shows two plots where DCV is used as a reference. There, we show the comparisons of the two optimum versions of the early terminated DC both for the model and the real executions on the C-3480. Figure 4.16.a shows case $R_{min} = 10$ (large diagonal dominance δ and low accuracy required ϵ) and Figure 4.16.b shows case $R_{min} = 100$ (small δ and high ϵ).

The difference between the early terminated and non-early terminated versions is substantial. Nevertheless, for large values of R_{min} the early termination does not pay off unless the matrices are large. Also, we can see in Figure 4.16 how the model for the early terminated versions of DC adapts to the actual executions on the C-3480. The error of the models with respect to the actual executions is of less than a 7%.

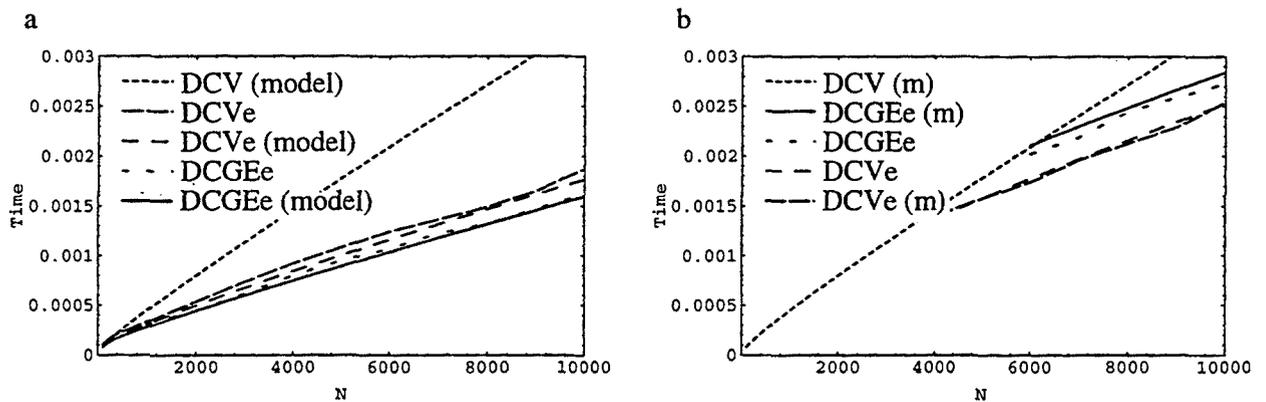


Figure 4.16. Comparison of the model of DC with the real executions on one processor of the C-3480 for the early terminated versions. a) $R_{min} = 10$, b) $R_{min} = 100$.

Note that both versions of DC behave differently for different values of R_{min} . While DCGEe works better for small values of R_{min} , DCVe is faster for large values of R_{min} . This is caused by the fact that for small values of R_{min} , the length of the vectors for DCVe is very small during stage 3, and for this reason the weight of the start-ups is considerable (note that for DCVe the start-ups of stage 3 depend strongly on the number of partitions and weakly on the value of R_{min}). For large values of R_{min} the vector length is large for both DCGEe and DCVe and, as a consequence, the weight of the start-ups is less significant for DCVe as the models show (the number of start ups of DCGEe depends strongly on the value of R_{min} and does not depend on the number of partitions).

4.4.3 The Overlapped Partitions Method

The implementation of OPM is quite simple, as we saw in chapter 3. Figure 4.17 shows a vector implementation of OPM in which each partition is solved with Gaussian Elimination.

The algorithm is divided into two parts, namely, the solution of the overlapping and the solution of the central part of the partitions. Loops with index j perform the traversing of the overlapping and the central part of a partition. Loops with index k perform the vectorization across partitions. Note the use of an auxiliary vector in the body of the loops that process the overlappings. This vector prevents the algorithm from modifying vector b during the processing of the overlapped equations. This is justified by two facts: first, vector b cannot be spoiled when solving the overlapped equations because the elements in the overlapping of partition k are used as elements in the central part of the partition for partition $k-1$; second, the only values of interest from the computation of the overlappings are in positions $R+1, 2R+1, 3R+1$, etc. Hence, only the last computed values of each overlapping have to be stored in vector b . These values correspond to the solution of the first equations of the central part of the partitions. The stride of the accesses to memory is R for this algorithm.

<pre> * Upper overlapping of partitions aux(1)=b(R-m+1) do j=R-m+1,R i=1 do k=j+1,N-m,R aux(i)=b(k)-a(k)*aux(i) i=i+1 enddo enddo i=1 do q=R+1,N-m,R b(q)=aux(i) i=i+1 enddo </pre>	<pre> * Central part of partitions do j=1,R do k=j+1,N,R b(k)=b(k)-a(k)*b(k-1) enddo enddo </pre>
---	---

Figure 4.17. Vector implementation of the Overlapped Partitions Method

Analysis of the assembly vector code of the Overlapped Partitions Method

The vector assembly pseudocode version of OPM is shown in Figure 4.18. In this case we assume that we do not apply strip mining and for this reason the number of partitions is limited by the size of the vector registers, MVL , as for the versions of DC.

The assembly pseudocode of OPM can be described as follows. Vector b is loaded before the iteration starts in order to reuse it through vector register $VR0$ along the different iterates of the loop. The difference between the code for the overlapping and the code for the central part of the partitions is in the store of the solution vector b . This means that the processing of the overlapping implies one store less per equation than the processing of the central part of the partitions.

Vector instructions performed outside the loop	VLOAD	$b(R-m+1), VR0$	Vector instruction performed outside the loop	VLOAD	$b(R+1), VR0$
Body of the loop of the overlapping	VLOAD	$b(k), VR2$	Body of the loop of the central part of the partition	VLOAD	$b(k), VR2$
	VLOAD	$a(k), VR1$		VLOAD	$a(k), VR1$
	VMUL	$VR0, VR1, VR3$		VMUL	$VR0, VR1, VR3$
	VSUB	$VR2, VR3, VR0$		VSUB	$VR2, VR3, VR0$
After the iteration finishes	VSTORE	$VR0, b(R+1)$		VSTORE	$VR0, b(k)$

Figure 4.18. Assembly vector pseudocode of the body of the loops of the Overlapped Partitions Method.

Scheduling of OPM for vector computers

As for the case of DC, it is possible to unroll some iterates of loop with index j of OPM to save some chimes to the execution of the algorithm. This can be done in a similar way to that shown in Figure 4.13 for DC. Here, we determine the amount of chimes performed as a function of the number of unrolled iterates.

The number of chimes for the partially unrolled versions and the non-unrolled versions of OPM are shown in Table 4.9. There, we show the number of chimes for the overlapping and the central part of the partitions. Also, we show the number of chimes per iterate given that we unroll U iterates.

		type 1	type 2	type 3	type 4	type 5
Chimes, non-unrolled	Overlapping	3	3	2	2	1
	Central	3	3	2	2	1
Chimes, unrolled $U > 2$	Overlapping	$Fo_1 = 2U + 1$	$Fo_2 = 2U + 1$	$Fo_3 = 2$	$Fo_4 = 2$	$Fo_5 = 1$
	Central	$Fc_1 = 3$	$Fc_2 = 2U + 1$	$Fc_3 = 2$	$Fc_4 = 2$	$Fc_5 = 1$
Chimes per iterate	Overlapping	$2U + 1/U$	$2U + 1/U$	2	2	1
	Central	3	$2U + 1/U$	2	2	1

Table 4.9. Number of chimes performed by different versions of OPM on the architectures studied.

A general model for OPM

The model for OPM is very simple. The number of chimes for U unrolled iterates of the overlapping is Fo_i . We process a total of m/U sets of unrolled iterates per partition (assuming that m is divisible by U). The vector length is $P^{(0)} - 1$ which is the number of partitions with overlapping. The amount of start-ups is $4m$.

The number of chimes for U unrolled iterates of the central parts of the partitions is Fc_i . We process a total of $(N/P^{(0)} - 1)/U$ sets of unrolled iterates per partition (assuming that $N/P^{(0)} - 1$ is divisible by U). The vector length in this case is $P^{(0)}$ and the amount of start-ups is equivalent to $5(N/P^{(0)} - 1)$.

The execution time model of OPM is, then

$$\begin{aligned}
 T_{OPM} &= 5(N/P^{(0)} - 1) \cdot T_{SU} + (N - P^{(0)}) \cdot Fc_i \cdot T_V + \\
 & 4m \cdot T_{SU} + m(P^{(0)} - 1) \cdot Fo_i \cdot T_V = \\
 & = [5(N/P^{(0)} - 1) + 4m] \cdot T_{SU} + [(N - P^{(0)}) \cdot Fc_i + m(P^{(0)} - 1) \cdot Fo_i] \cdot T_V.
 \end{aligned}$$

And the optimum number of partitions is

$$P^{(0)} = \sqrt{5NT_{SU} / [(Fo_i m - Fc_i) T_V]}.$$

Finding an optimum value of the partial unrolling U

We have computed the optimum number of partially unrolled iterates in the same way as we did for DC supposing that $mP^{(0)} = N$. The optimum values of U for the architectures studied are shown in Table 4.10.

DC	type 1	type 2	type 3	type 4	type 5
<i>U overlapping</i>	5	5	1	1	1
<i>U central</i>	1	5	1	1	1

Table 4.10. Optimum value of U for different types of architectures for OPM assuming $\phi = 0.03$.

Now, we evaluate the relative importance of choosing the value of U in the context of OPM for architecture type 1. Figure 4.19.a shows a comparison of the modelled execution times for two different values of m ($m = 10$ and $m = 100$) and U ($U = 1, 2, 3$). Figure 4.19.b shows the same comparison for the real executions on the C-3480 and $U = 1, 2$ assuming the optimum practical values of $P^{(0)}$ computed below. There, we can see that the gain between $U = 1$ and $U = 2$ for the models is significative although for the real executions this gain can hardly be noticed, specially for the case $m = 10$. Although the difference between $U = 1$ and $U = 2$ is small, we choose $U = 2$ for the real measurements.

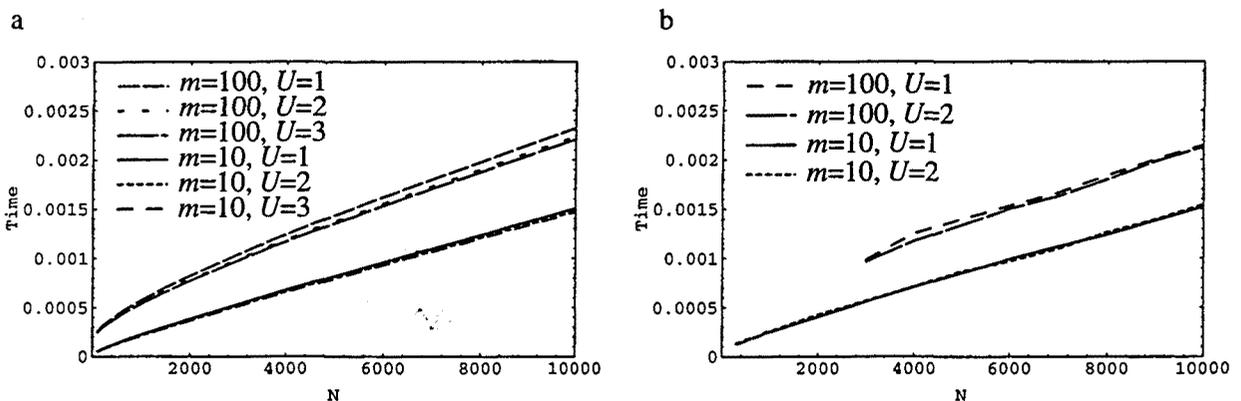


Figure 4.19. Modelled (a) and real (b) execution times of OPM for different values of m and U .

Note that the gains achieved in Figure 4.19.a and 4.19.b do not correspond to the prospects of Table 4.10. The reason for this is that in Table 4.10 we supposed $mP^{(0)} = N$ which is not the case in general. Lower values of U are enough as we have seen in Figure 4.19.

Optimum number of partitions for OPM

In order to compute the practical optimum values of $P^{(0)}$ for OPM on the C-3480 we have fitted the models of the method as we did for DC. The model that we fit is

$$T_{OPM} = K_1 N + K_2 P^{(0)} + K_3 \frac{N}{P^{(0)}} + K_4 m + K_5 m P^{(0)} + K_6$$

and the corresponding values of the constants after the fits are

$$\begin{aligned} \text{OPM} \rightarrow K_1 &= 9.3 \cdot 10^{-8} & K_2 &= 1.7 \cdot 10^{-6} & K_3 &= 2.9 \cdot 10^{-6} & K_4 &= 1.6 \cdot 10^{-6} \\ & & K_5 &= 9.3 \cdot 10^{-8} & K_6 &= -2.4 \cdot 10^{-5} \end{aligned}$$

The mean quadratic error of the fit is less than a 7%. With this, the optimum value of $P^{(0)}$ for OPM is $P^{(0)} \cong \sqrt{(87.4N) / (51.6 + 2.8m)}$ and the theoretical optimum values of $P^{(0)}$ are $P^{(0)} \cong \sqrt{(100N) / (6 + 5m)}$ which differ significantly from the practical values on the C-3480. The reason for this is in the fact that the execution time is not considerably sensitive to fluctuations of $P^{(0)}$ around the optimum like DCGE.

Analyzing the model for different computers

Now, we validate the execution time model of OPM with real executions on the C-3480. Figure 4.20 shows a comparison of the models and the real measurements of OPM for the case $U = 2$. The execution times of OPM are compared to the model of 9-CR to give a measure of the relative gain of the method. As we can see, the gain is considerable for small values of m and can be significative for large matrices for cases with large m . The difference between the models and the real execution is smaller than an 8% for the worst case which validates the execution time model of OPM.

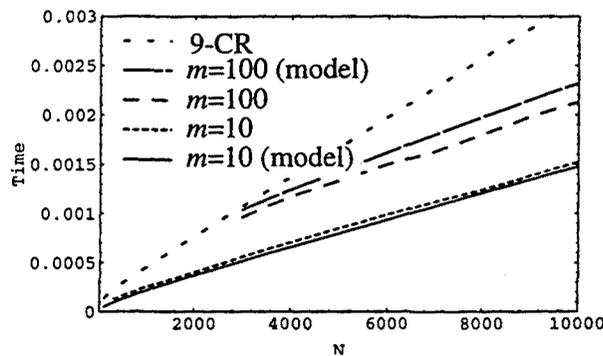


Figure 4.20. Validation of the execution time model of OPM and comparison with the model of 9-CR.

4.5 Comparison of the models for different types of computers

In this section, we present comparisons of the modelled execution times of the methods studied up to now. For each of the five architecture types and all the versions of the algorithms we present two types of plots.

On one side, we show the modelled execution time of the methods as a function of the size of the problem N . With these plots we compare the fastest non-early terminated versions of DC and R -CR and, the early terminated versions of DC and OPM. For DC and OPM we compare two cases for fixed values of R_{min} and m ($R_{min} = m = 10$ and $R_{min} = m = 100$) in order to give an idea of how the methods compare for large and for weak diagonal dominances. The value of N in those plots ranges from 1 to 10,000.

On the other side, we show a comparison of the absolute suitability of the methods (early and non-early terminated) for different values of δ and N and for two different values of ε ($\varepsilon = 10^{-7}$ and $\varepsilon = 10^{-16}$). So, for the pairs (δ, N) in the range $(1.05 < \delta < 2, 10 < 2500)$ we show the fastest algorithm and the percentage of improvement relative to the second best method. This comparison is intended as an aid for building libraries given that a problem is defined by its pair (δ, N) and the maximum absolute error allowed to the solution of the system ε .

One important thing is that Gaussian elimination is not plotted in the comparisons shown here. The reason for this is that the model of Gaussian elimination gives higher execution times than the rest of the models for any value of N . Nevertheless, we must say that Gaussian elimination is faster than the other methods for small systems on real computers. For instance, on the C-3480, Gaussian elimination is the fastest method for systems of order smaller than 160. This better performance is caused by the little influence of control instructions on Gaussian elimination and the optimum use of scalar data caches.

Figures 4.21 and 4.22 show the comparison of the methods on one vector processor with one bidirectional path to memory that only allows output chaining (architecture type 1). OPM is considerably faster than the other methods both for large and small values of R_{min} and m . Also note that for non-strictly diagonal dominant systems 9-CR is faster than DCV. For this reason we do not plot DCV in Figure 4.21.b.

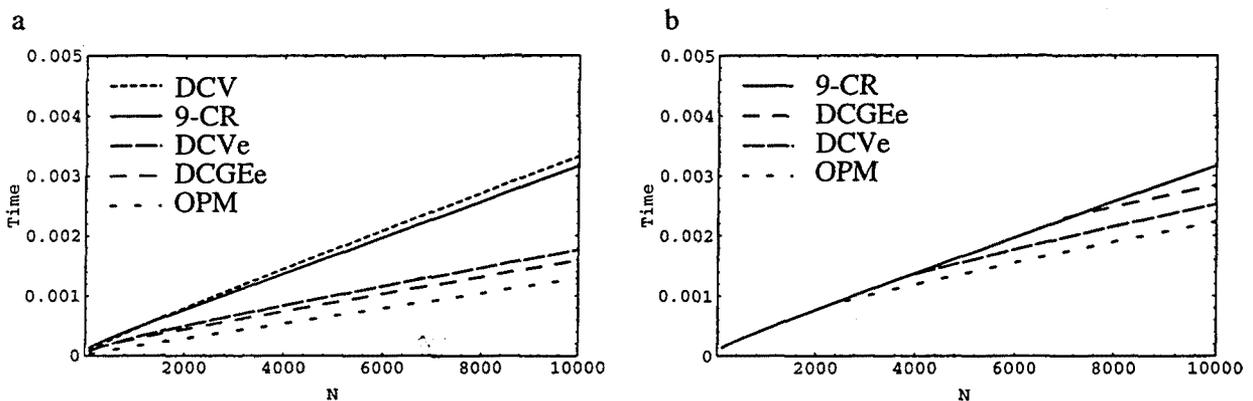


Figure 4.21. Comparison of 9-CR, DCV, DCGEe, DCVe and OPM for architecture type 1.

a) $R_{min} = m = 10$. b) $R_{min} = m = 100$.

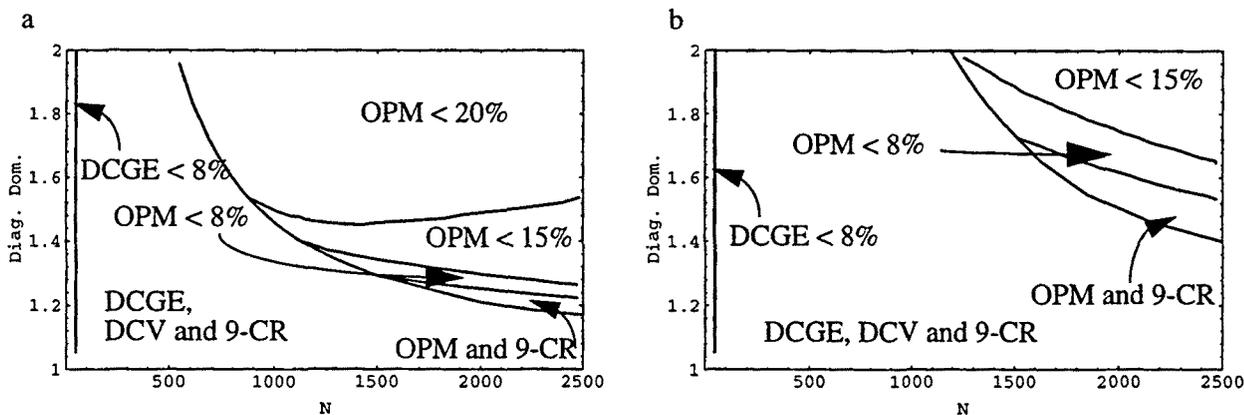


Figure 4.22. Absolute suitability of solvers on architecture type 1 for different values of N and δ . The maximum differences allowed between methods are 8%, 15% and 20%. a) $\epsilon = 10^{-7}$. b) $\epsilon = 10^{-16}$.

Figures 4.22.a and 4.22.b show that DCGE is faster than the other methods for small systems of equations in more than an 8%. Those Figures show that there is a zone where both DCGE, DCV and 9-CR have an execution time that differs in less than an 8% and another zone where both OPM and 9-CR have similar execution times which are considerably lower than rest of the methods. Finally, the plots show contour lines that give an idea of the difference in time between OPM and the second best method.

Figures 4.23 and 4.24 show the comparison of the methods for the architecture with two unidirectional paths to memory that allow only output chaining (architecture type 2). Note in Figure 4.23 that for large systems and large diagonal dominances ($R_{min} = m = 10$) DCGEe has a similar execution time to OPM. Nevertheless, for systems with weak diagonal dominance, OPM is faster as it can also be seen in Figure 4.23.b. Figure 4.24 is similar to Figure 4.22 except for the fact that OPM is a bit slower. Also, one important point to notice in Figures 4.24.a and 4.24.b is that there is an area where 9-CR is faster than the rest of the methods in more than an 8%. This is for systems with weak diagonal dominance where the early terminated methods and OPM do not apply.

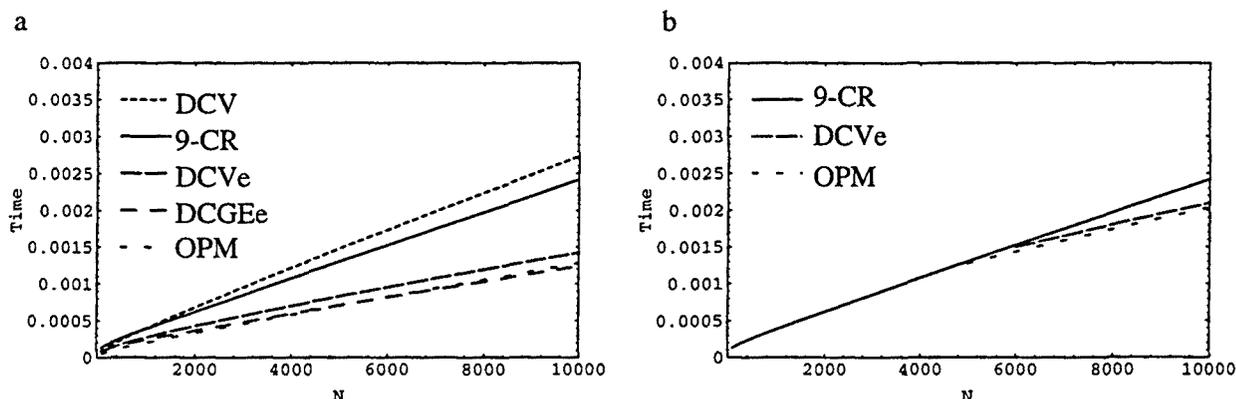


Figure 4.23. Comparison of 9-CR, DCV, DCGEe, DCVe and OPM for architecture type 2. a) $R_{min} = m = 10$. b) $R_{min} = m = 100$.

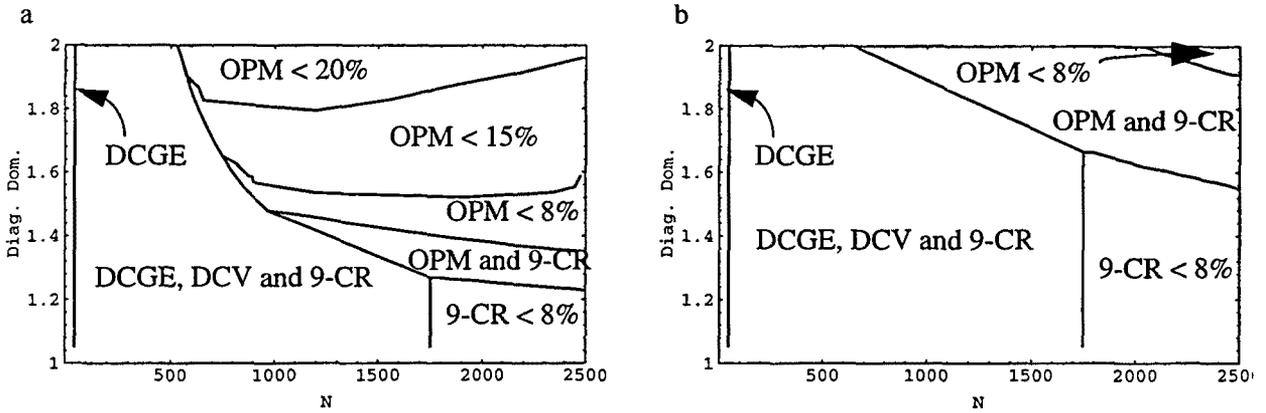


Figure 4.24. Absolute suitability of solvers on architecture type 2 for different values of N and δ . The maximum differences allowed between methods are 8%, 15% and 20%. a) $\epsilon = 10^{-7}$. b) $\epsilon = 10^{-16}$.

Figures 4.25 and 4.26 show the comparison of the methods for vector processors with two unidirectional paths to memory that allow input and output chaining (architecture type 3). The important point to note here is that DCV shows to be the fastest method for systems of order between 70 and 230 approximately. The rest of the methods behave similarly to the methods in the previous case and do not need further comment.

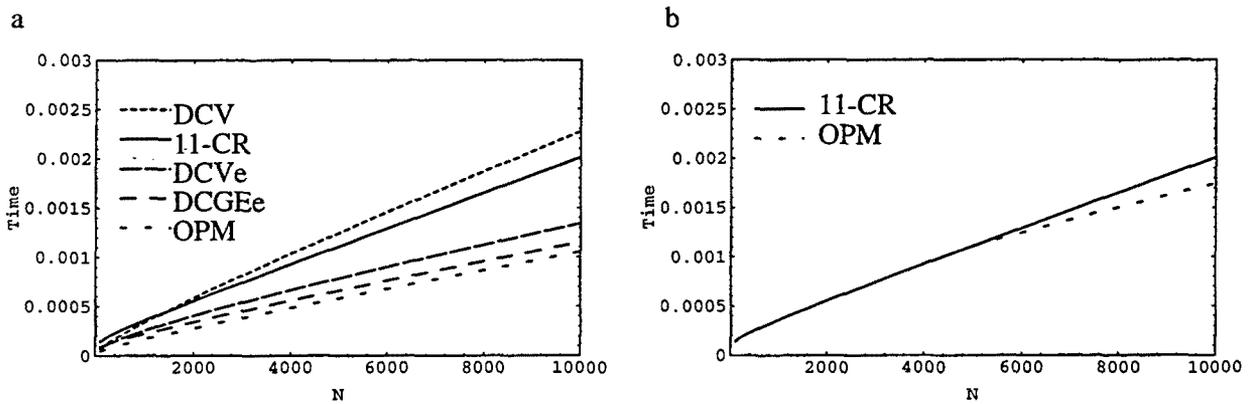


Figure 4.25. Comparison of 11-CR, DCV, DCGEe, DCVe and OPM for architecture type 3. a) $R_{min} = m = 10$. b) $R_{min} = m = 100$.

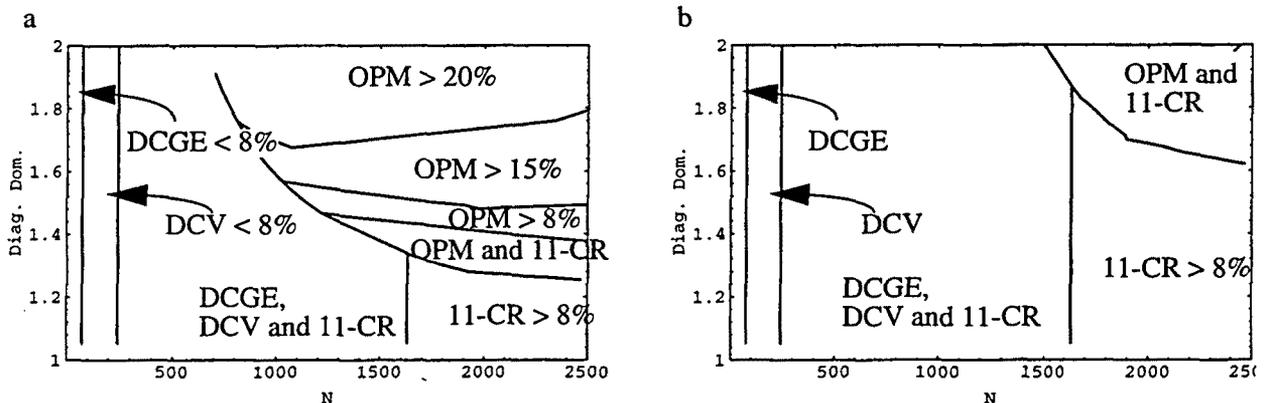


Figure 4.26. Absolute suitability of solvers on architecture type 3 for different values of N and δ . The maximum differences allowed between methods are 8%, 15% and 20%. a) $\epsilon = 10^{-7}$. b) $\epsilon = 10^{-16}$.

Figures 4.27 and 4.28 show the plots of the execution time for the type of processor with two bidirectional paths to memory that allow input and output chaining (architecture type 4). There are not many differences between these plots and those for the cases shown above for processors with two paths to memory. The only important difference in this case is that OPM is slower than DCGEe for large diagonal dominances in the case of large matrices as we can see in Figure 4.27.a. This tendency can also be observed in Figure 4.28.a where the contour lines of OPM show that for large systems with large diagonal dominances the method tends to have a lower advantage compared to the other methods. It is important to note that, as the resources increase (number of paths to memory and chaining capability), the non-early terminated variants of the methods behave more similarly to the early terminated ones.

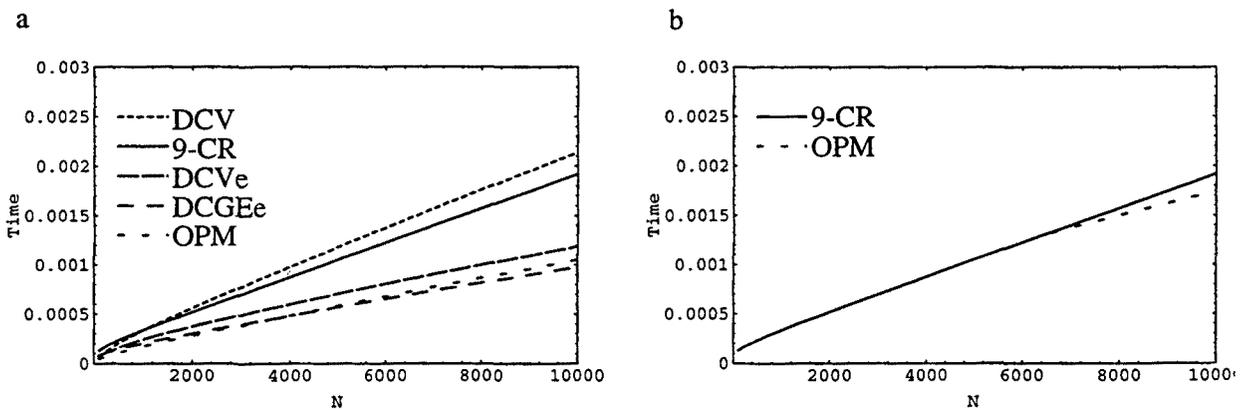


Figure 4.27. Comparison of 9-CR, DCV, DCGEe, DCVe and OPM for architecture type 4. a) $R_{min} = m = 10$. b) $R_{min} = m = 100$.

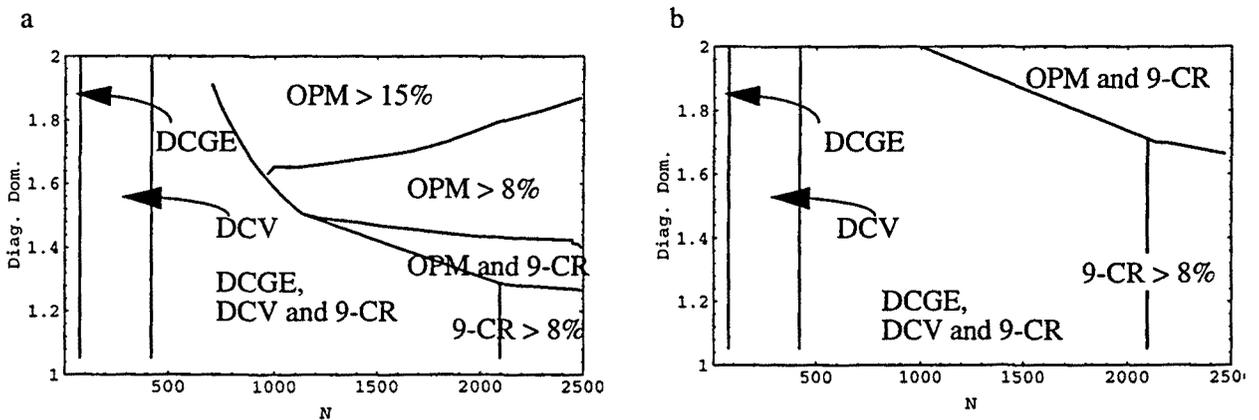


Figure 4.28. Absolute suitability of solvers on architecture type 4 for different values of N and δ . The maximum differences allowed between methods are 8%, 15% and 20%. a) $\epsilon = 10^{-7}$. b) $\epsilon = 10^{-16}$.

Figures 4.29 and 4.30 show the comparison of the algorithms on the type of processor with 2 load and 1 store paths to memory that allow input and output chaining (architecture type 5). Figure 4.29.a shows that, in general OPM is considerably faster than the rest of the methods in the case of large diagonal dominances. Also, for the case of weak diagonal dominances and large matrices $N > 2600$ (Figure 4.29.b), OPM is also

competitive compared to the early terminated versions of the algorithms. Note that the behaviour of OPM compared to the rest of the methods on this type of computer is considerably better than on processors with 2 paths to memory for extreme cases. Nevertheless, for not so extreme cases, the rest of methods play an important role as we can see in Figure 4.30.

Figure 4.30 shows that for small systems DCV is faster than the other methods in an 8% (systems of order smaller than approximately 700). It is also important to notice that DCVe has a small area for which it is faster than the other methods which is a difference with the other types of computers. Also, DCGE is not faster than the other methods for small systems in contrast with the other types of processors.

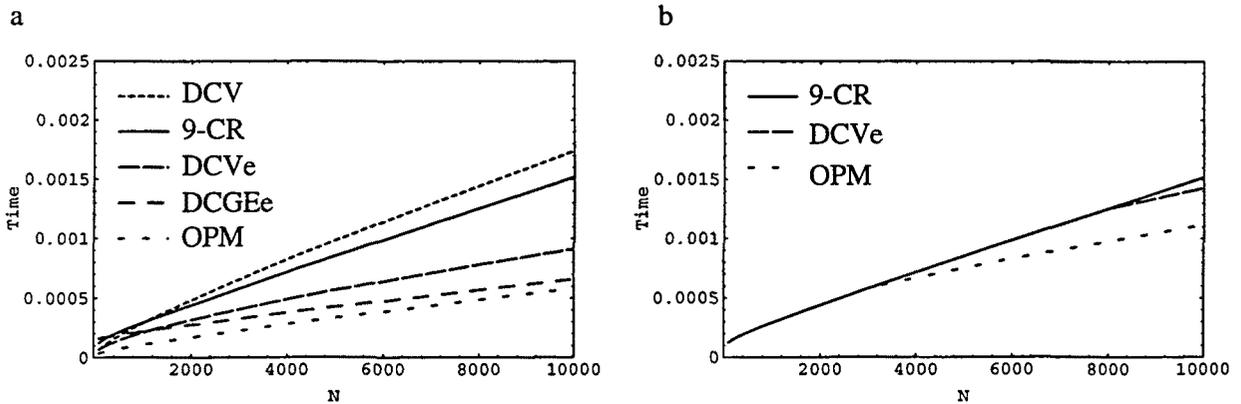


Figure 4.29. Comparison of 9-CR, DCV, DCGEe, DCVe and OPM for architecture type 5. a) $R_{min} = m = 10$. b) $R_{min} = m = 100$.

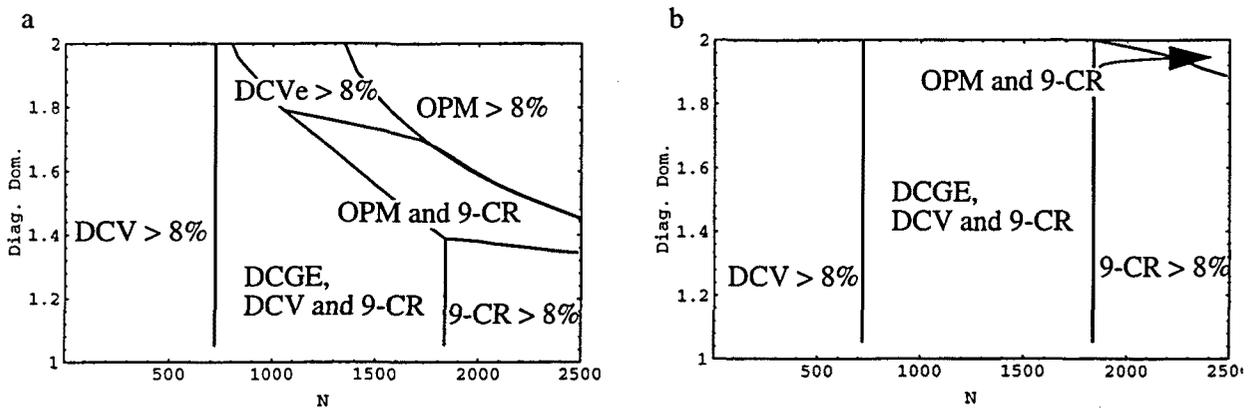


Figure 4.30. Absolute suitability of solvers on architecture type 5 for different values of N and δ . The maximum differences allowed between methods are 8%, 15% and 20%. a) $\epsilon = 10^{-7}$. b) $\epsilon = 10^{-16}$.

Chapter 5: Bidiagonal solvers on parallel computers

In this chapter we analyze the relation between communication and computation in the parallel implementation of R-Cyclic Reduction, Divide and Conquer and the Overlapped Partitions Method for bidiagonal systems. We assume a message passing type of architecture and build models of the methods. Also, we compare the different methods given different significative values of the communication and computation times. Finally, we give an example of execution of the algorithms on the Paramid 16/860SYS parallel computer using the PVM message passing tool.

5.1 Introduction

During the chapter, we first describe the work performed in the field of parallel bidiagonal solvers. A study of Divide and Conquer (that has been previously analyzed for parallel computers), *R*-Cyclic Reduction and the Overlapped Partitions Method follows. In the study, models are built to understand the behaviour of the methods on parallel computers with different communication and computation times. With this study and the understanding of the unified algorithm of chapter 2, we propose a new variant of *R*-CR that can be applied successfully on parallel computers in some cases. The execution time models of the algorithms are compared for different values of communication and computation times. Finally, we describe the Pyramid 16/860SYS and its implementation of PVM and show an example of the execution of Divide and Conquer on it.

5.2 Previous work

Now we describe two papers that evaluate the parallel implementation of bidiagonal solvers. Those papers were written by Van der Vorst and study the use of Divide and Conquer and analyze it for different parallel computers [Vand88, Vand89]. In both papers the author argues that two versions of DC can be used. Nevertheless, as the author says, one of those versions is not competitive because it requires more communication than the other. In particular, the two versions are equivalent to the two vector versions for DC studied in chapter 4. Obviously, the version corresponding to that in which vectorization is found within partitions during stage 3 requires a reordering of the data in such a way that the elements that were in one processor during stage 1 have to be distributed during stage 3. This requires a considerable amount of unnecessary effort because stage 3 can be performed in parallel without changing the data distribution. An observation of Van der Vorst in those papers is that the distribution of a number of partitions larger than one per processor degrades the performance of the algorithm. This is clear because the larger the number of partitions, the larger is the reduced system to be solved sequentially.

5.3 Target architecture

Parallel computers have been classified in different ways. For instance, Flynn's taxonomy determines the way in which instructions and data are related. There are four possible types of computer with this classification but the two most relevant in terms of parallel computation are SIMD (Single Instruction stream/Multiple Data stream) and MIMD (Multiple Instruction stream/Multiple Data stream) which have been widely used to define different models of parallel computation and are described in detail in [HoJe88]. This classification implies that SIMD computers require a unique central control unit that synchronizes the operation of the different processing units. MIMD computers require a distributed control so that the different processing units can work asynchronously.

Another type of classification determines the way in which memory and processors are interconnected: in a Shared Memory Multiprocessor (SMM) all processors can see a unique address space so that the same address for two different processors leads to the same memory position; in a Distributed Memory Multiprocessor (DMM) each processor can only see its own address space so that the same address for two different processors leads to different memory positions. This classification implies that while a processor of a SMM can communicate with another processor in the same computer by means of storing data in a certain memory position, a processor of a DMM needs to perform communication primitives (message passing) in order to have access to data stored in another processor's memory. Nevertheless, it is possible to have DMMs with virtual shared memory that allow the use of a single address space for all processors and SMMs with virtual disjoint address spaces for each processor. This implies that we can use virtual message passing in some SMMs environments and virtual communication through memory in some DMMs environments. We do not observe this two last approaches here.

In this chapter we analyze the use of MIMD computers with Distributed Memory organization for solving our problem. A simple scheme of a computer of this type is shown in Figure 5.1. Here, we suppose that each processor is scalar and the interconnection network topology is a crossbar switch. Each processor has a certain number larger than two of parallel communication channels connected to the crossbar. That is, one processor can communicate with a number of processors equal to the number of channels at the same time. Groups of different processors can communicate in parallel given that they use different channels.

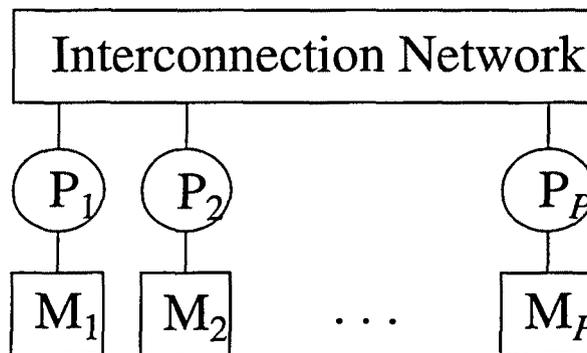


Figure 5.1. Scheme of a message passing parallel computer.

Within this organization we assume the need to use message passing primitives to communicate data. The communication primitives can be of either of the following two types as we saw in chapter 2

- send (ξ, \dots, μ, π) , which sends data items ξ, \dots, μ to processor π .
- receive (ξ, \dots, μ, π) , which receives data items ξ, \dots, μ from processor π .

These communication primitives are asynchronous with blocking receive and non-blocking send. That is, once the send primitive has performed its start-up, the processor can continue performing other work. For a receive primitive, the processor cannot continue until the complete message has been received.

Modelling the execution time of a parallel algorithm

A parallel algorithm is formed by different processes that run on different processors. The model of a parallel algorithm is determined by its critical path which can be found with the help of its time diagram. A time diagram of a hypothetical parallel algorithm and the critical path of the algorithm is shown in Figure 5.2. The horizontal time axis of the diagram denotes the different events and the order in which they happen in a certain time scale. The shadowed boxes denote when and how long each process is working for. The black arrows denote when a process sends data to another process. Finally, the absence of a shadowed box on the time axis of a processor means that the processor is idle waiting for a message. The critical path is determined by the striped line. Note that the critical path is a continuous line (uninterrupted execution and communication) that determines the parallel execution time. This type of time diagram will be used along the chapter to determine the critical path of the different algorithms studied.

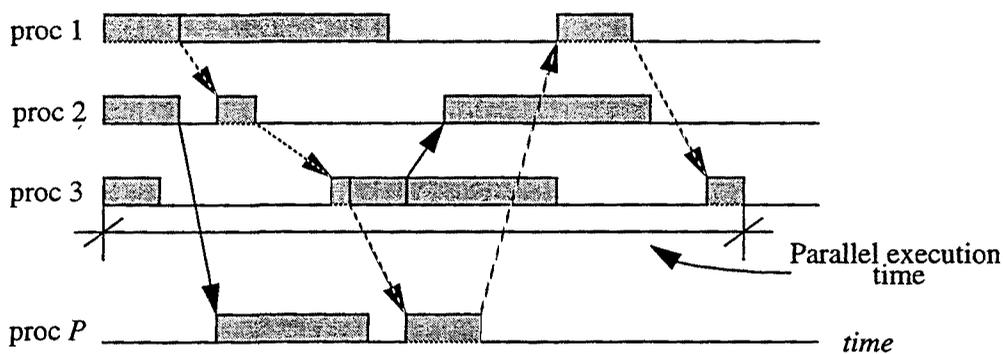


Figure 5.2. Time diagram and critical path of a hypothetical parallel algorithm.

Therefore, the model of a parallel algorithm is equal to the addition of all the computation and communication times along the critical path. This leads to a formula of the following type

$$T_{par} = T_{comp} + T_{comm}$$

where T_{comp} is the time spent in computations and depends on the characteristics of the sequential algorithm executed in each node and on the architectural characteristics, i.e. superscalar ability/vector ability, number and size of cache levels, etc.. Also, T_{comm} is the time spent in communication and depends on the factors that determine the speed of the interconnection network.

Here, we use a simple model of the execution of a code in a processing node where the arithmetic and memory instructions have a similar execution cost of T_S seconds. Thus $T_{comp} = K_S \cdot T_S$ where K_S is the number of arithmetic and memory instructions of the code. Also, we use a simple model of communication in which the time to perform a communication is $T_{SU} + K_i \cdot T_E$. where T_{SU} is the start-up time, K_i is the length of the message and T_E is the time to send an element of the message. So, we are going to use the following model of execution time for the parallel programs analyzed

$$T_{par} = K_S \cdot T_S + K_{SU} \cdot T_{SU} + K_E \cdot T_E$$

where K_{SU} is the total number of messages performed and K_E the total number of elements sent by the K_{SU} messages performed. This is the model that we are going to use in the following sections.

5.4 The parallel algorithms

In this section we describe the parallel implementations of the algorithms and build their execution time models. For the description of the algorithms, we preserve the notation of chapter 2.

5.4.1 Divide and Conquer

Divide and Conquer is a method that suits parallel computers because the number of partitions formed for its implementation can be equal to the number of processors that work in parallel. In this way parallelism is exploited at its best and only a little amount of communication is required to solve the reduced system. Figure 5.3 shows the versions of DC for diagonal dominant and strictly diagonal dominant systems using the communication primitives mentioned above. Here, we suppose that each partition of the algorithm is assigned to one processor.

<pre> do r = 1, R - 1 b_[r+1] = b_[r+1] - b_[r] a_[r+1] a_[r+1] = -a_[r] a_[r+1] enddo if p > 1 then receive (aux, taskid[p - 1]) b_[R] = b_[R] - aux · a_[R] endif if p < P then send (b_[R], taskid[p + 1]) endif do r = 1, R - 1 b_[r] = b_[r] - aux · a_[r] enddo </pre>	<pre> do r = 1, R_{min} - 1 b_[r+1] = b_[r+1] - b_[r] a_[r+1] a_[r+1] = -a_[r] a_[r+1] enddo do r = R_{min}, R - 1 b_[r+1] = b_[r+1] - b_[r] a_[r+1] enddo if p < P then send (b_[R], taskid[p + 1]) endif if p > 1 then receive (aux, taskid[p - 1]) endif do r = 1, R_{min} b_[r] = b_[r] - aux · a_[r] enddo </pre>
(a)	(b)

Figure 5.3. Parallel implementation of Divide and Conquer for a message passing parallel computer. a) Non-strictly diagonal dominant systems. b) Strictly diagonal dominant systems.

Figure 5.3.a shows the parallel program for non-strictly diagonal dominant systems of equations. After receiving the system to be solved, process p starts performing the first stage of the algorithm without

communication. With the solution to the first phase, the solution of the reduced system can be computed with communication. We choose to solve of the reduced system in a distributed way. Finally, stage three is performed without communication.

Figure 5.3.b shows the algorithm for the early terminated version of DC. Note that stage 1 is divided into two loops. Also, the solution of the reduced system is not performed. Nevertheless, some communication is needed to allow each processor have the solution to the last equation of the previous partition. These communications can be performed in parallel for the model of parallel computer described above. Finally, during stage 3 it is only necessary to perform some work on part of the equations of each partition.

A model for DC

In order to build the model of DC we find its critical path. Figure 5.4 shows a time diagram with the processor execution time and communication for a problem solved with P processors. If we suppose that all processors start at the same time, the critical path shown in Figure 5.4 is determined by the execution time of stage 1 on processor 1 plus the execution of stage 3 on processor P $t_1 + t_4$ plus the propagation of the solution of the reduced system, $(P - 1) \cdot (t_2 + t_3)$.

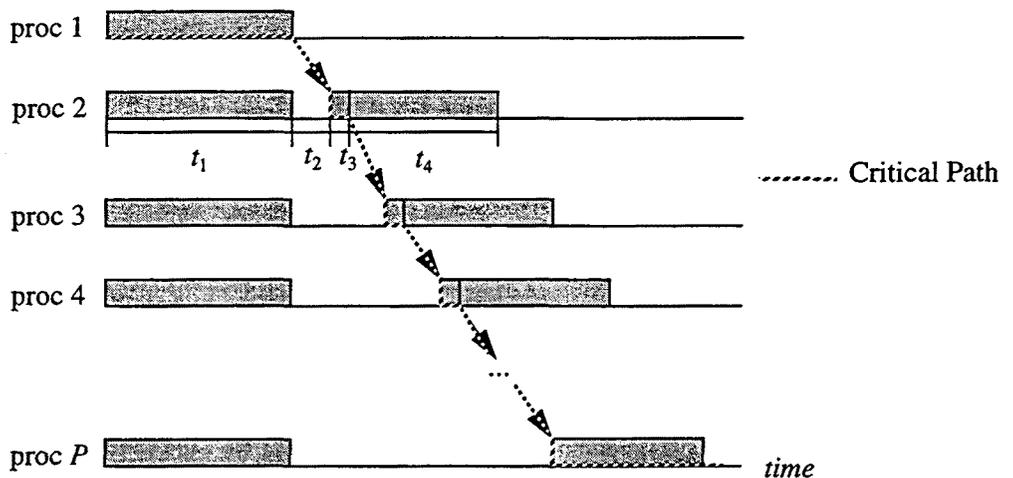


Figure 5.4. Time diagram and critical path of the execution of DC. t_1 , execution time of stage 1; t_2 , communication time for the solution of one equation of the reduced system; t_3 , computation time to solve the equation of the reduced that has processor p and, t_4 is the execution time of stage 4.

Given the critical path of Figure 5.4 and the algorithm of Figure 5.3.a, the model is as follows. The amount of computations during stages 1 and 3 is equivalent to $13(N/P - 1)$ per processor. This is because a number of 4 loads, 3 stores and 6 arithmetic operations are performed per iterate if we reuse variables through registers. The solution of the reduced system implies that we perform a total of $P - 1$ communications with messages of size 1 element. Also, a total of 6 computations are performed to solve each equation of the reduced system (3 loads, 1 store and 2 arithmetic operations). So, the total execution time can be modelled very simply by

$$(5.1) \quad [13(N/P - 1) + 6(P - 1)]T_S + (P - 1)T_{SU} + (P - 1)T_E$$

where the optimum number of processors is

$$P = \left\lceil \sqrt{\frac{13T_S N}{6T_S + T_E + T_{SU}}} \right\rceil$$

from which we deduce that given a specific architecture (T_S, T_{SU}, T_E) it is not worthwhile to use more than P processors, although we have them.

Let us find a model for the early terminated DC. This early termination implies that the reduced system is not solved. In this case, the time diagram is much simpler because during the phase in which messages are passed along the critical path, all processors can communicate at the same time given the model of computer that we assume here. This is shown in the time diagram of Figure 5.6.

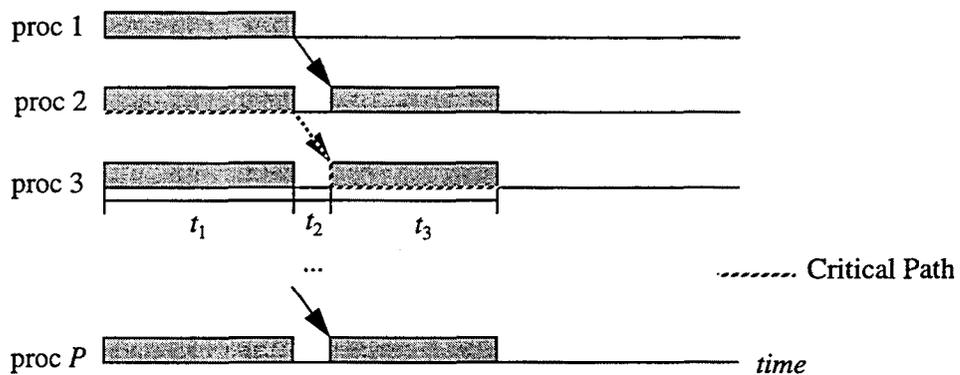


Figure 5.5. Time diagram and critical path of the early terminated DC. t_1 , execution time of stage 1; t_2 , communication time for the solution of the reduced system and; t_3 , execution time of stage 1.

Given the critical path of Figure 5.5 and the algorithm of Figure 5.3.b, we can build the model of the early terminated DC. The amount of computations during stages 1 and 3 is $13(R_{min} - 1) + 5(R - R_{min})$. Also, 1 communication is performed in the critical path. The execution model of DC is then

$$(5.2) \quad [13(R_{min} - 1) + 5(R - R_{min})] T_S + T_{SU} + T_E$$

where the optimum number of processors is determined by

$$P = \frac{N}{R} \leq \frac{N}{R_{min}}$$

which means that the size of the partitions R has to be larger than R_{min} .

5.4.2 R-Cyclic Reduction

We consider two implementations of R -CR on parallel computers: a) An implementation in which one initial partition is distributed per processor $N/P^{(0)} = R$, b) an implementation in which more than one initial partition is distributed per processor $N/P^{(0)} \gg R$. For the analysis of those two cases we introduce

the data transport and computation graphs. We show one of those graphs for each of the two algorithms analyzed in Figures 5.7 and 5.8. The boxes in the data transport and computation graphs mean “processor busy” rather than “amount of work” that is what they mean in the time diagrams. Also, the arrows mean communication rather than any measure of the time spent to perform a communication. The aim of the data transport and computation graphs is not to obtain the parallel execution time (we use the time diagrams for this purpose) but to display the amount of communications for each approach a) and b).

We first comment on the implementation of R -CR in which one partition is assigned per processor during step 0. To our knowledge, this approach for parallel computers is new in the literature. For this algorithm we only discuss a parallel version in which two steps plus the solution of the reduced system of R -CR are performed although versions with more steps could be implemented. A possible algorithm for a two step version of R -CR is shown in Figure 5.6 and a data transport and computation graph for an example is shown in Figure 5.7. We explain this algorithm below.

```

 $pclus = P^{(0)} / P^{(1)}$ 
(*step 0 forward*)
do  $r = 1, R - 1$ 
     $b_{[r+1]} = b_{[r+1]} - b_{[r]} a_{[r+1]}$ 
     $a_{[r+1]} = -a_{[r]} a_{[r+1]}$ 
enddo
(*step 1 forward*)
if  $p \bmod pclus \neq 1$  then
    receive ( $baux, taskid[p - 1]$ )
    receive ( $aaux, taskid[p - 1]$ )
     $b1 = b_{[R]} - baux \cdot a_{[R]}$ 
     $a1 = -aaux \cdot a_{[R]}$ 
endif
if  $p \bmod pclus \neq 0$  then
    send ( $b1, taskid[p + 1]$ )
    send ( $a1, taskid[p + 1]$ )
endif
(*reduced system *)
if  $p \bmod pclus = 0$  and  $p > pclus$  then
    receive ( $aux, taskid[p - pclus]$ )
     $b_{[R]} = b_{[R]} - aux \cdot a_{[R]}$ 
endif
if  $p \bmod pclus = 0$  and  $p < P^{(0)}$  then
    send ( $b_{[R]}, taskid[p + pclus]$ )
endif

(*step 1 backward*)
if  $p \bmod pclus = 0$  and  $p < P^{(0)}$  then
    send ( $b_{[R]}, taskid[p + 1]$ )
endif
if  $p \bmod pclus = 1$  and  $p > pclus$  then
    receive ( $aux, taskid[p - 1]$ )
endif
if  $p \bmod pclus = 0$  then
    do  $i = 1, pclus - 2$ 
        send ( $b_{[R]}, taskid[p - i]$ )
    enddo
endif
if  $p \bmod pclus > 1$  then
    receive ( $aux,$ 
         $taskid[p + pclus - (p \bmod pclus)]$ )
     $baux = baux - aux \cdot aaux$ 
endif
(*step 0 backward*)
do  $r = 1, R$ 
     $b_{[r]} = b_{[r]} - baux \cdot a_{[r]}$ 
enddo

```

Figure 5.6. Parallel implementation of a two step version of R -Cyclic Reduction for a message passing parallel computer assuming that each partition is assigned to one processor.

If we have a certain number of processors $P^{(0)}$ ($P^{(0)} = 9$ for the example of Figure 5.7), the size of the partitions is $R = N/P^{(0)}$ (*step 0 forward* in Figure 5.6) and the reduced system formed during this step is of order $P^{(0)}$. This reduced system can be solved in parallel now. If we suppose that $P^{(1)}$ partitions are formed ($P^{(1)} = 3$ for the example of Figure 5.7), each partition can be processed by clusters of $P^{(0)}/P^{(1)}$ processors in cooperative form (*step 1 forward* in Figure 5.6). The reduced system formed during step 1 can be solved by $P^{(1)}$ non neighbouring processors in cooperative form again (*reduced system* in Figure 5.6). The implementation of the solution to this reduced system is like that for the solution to the reduced system in DC. Then, during the backward phase of step 1 (*step 1 backward* in Figure 5.6) the solution to the equations of the reduced system is distributed among the processors involved in the backward phase of step 1. Finally, the solutions to the rest of equations of the system are found in parallel during the backward phase of step 0 without any need for communication.

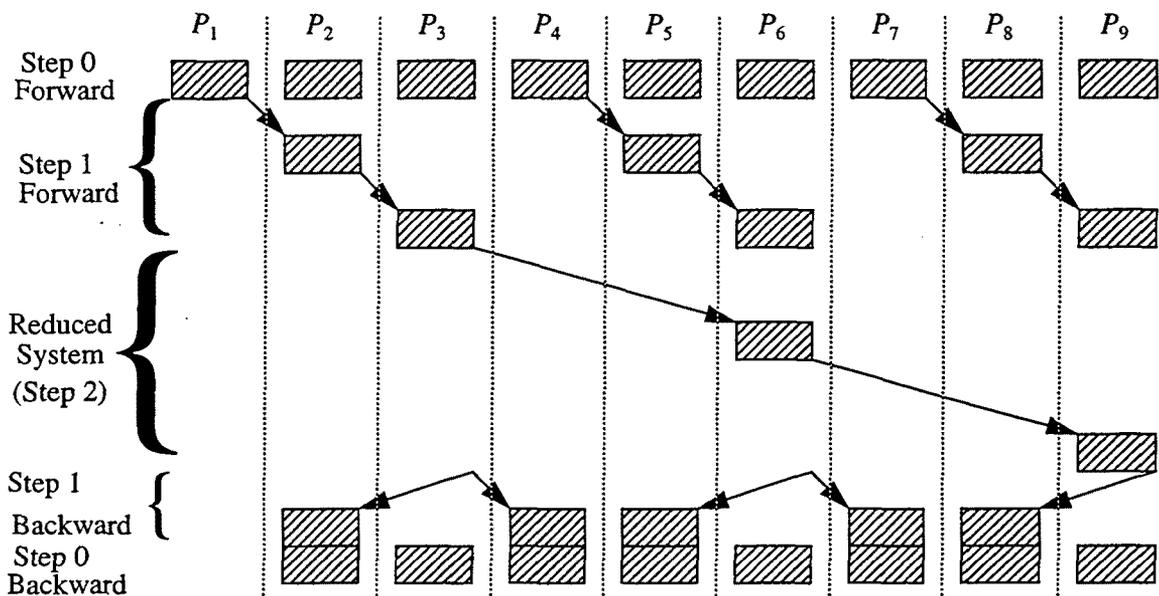


Figure 5.7. Data transport and computation graph of the algorithm of Figure 5.6 with $N/P^{(0)} = R$.

The algorithm in which more than one initial partition $N/P^{(0)} \gg R$ is assigned per processor implies more communication than the previous algorithm. We comment on this algorithm but we do not show an algorithm for it because this algorithm is not going to be considered. A data transport and computation graph of an example for that case is shown in Figure 5.8. In Figure 5.8, we can see that after each forward step it is necessary to communicate. This is because a processor may require some equations of a neighbouring partition to have a complete partition and be able to perform the following step. It must be said that the amount of computations performed during the forward steps 0, 1 and 2 of the example of Figure 5.8 is the same as that performed during the forward step 0 of the example in Figure 5.7 although the data transport

and communication graph does not reflect it. When only one equation per processor remains (after step 2 in the example of Figure 5.8), the reduced system has to be solved. This can be done as for the case of DC or as for the case of the R -CR explained above. During the backward steps, it is necessary to perform a similar amount of communications and the amount of computations are equal to the backward steps of the example in Figure 5.7. So, in general, we can say that this algorithm implies more communication than the previous one and for this reason we do not consider this partitioning of the data.

Now we give a detailed account of the number of communications performed during the forward steps of the algorithm. The total number of communication phases during the forward phase of the algorithm may be of up to $\log_R(N/P^{(0)})$. A lower number of steps may be necessary though. If the number of equations assigned per processor is equal to a power q of the size of the partition $R(N/P^{(0)} = R^q)$ then q steps are performed in parallel by all processors and the number of communication phases reduces to 0 during the forward phase. For the example of Figure 5.8, if $N/P^{(0)} = R^3$ there would be no communication during the forward phase. If the number of equations assigned per processor is equal to $N/P^{(0)} = \alpha R^q$ for $\alpha \neq R^q$ then the number of communication phases ranges from 1 to $\log_R(N/P^{(0)}) - 1$.

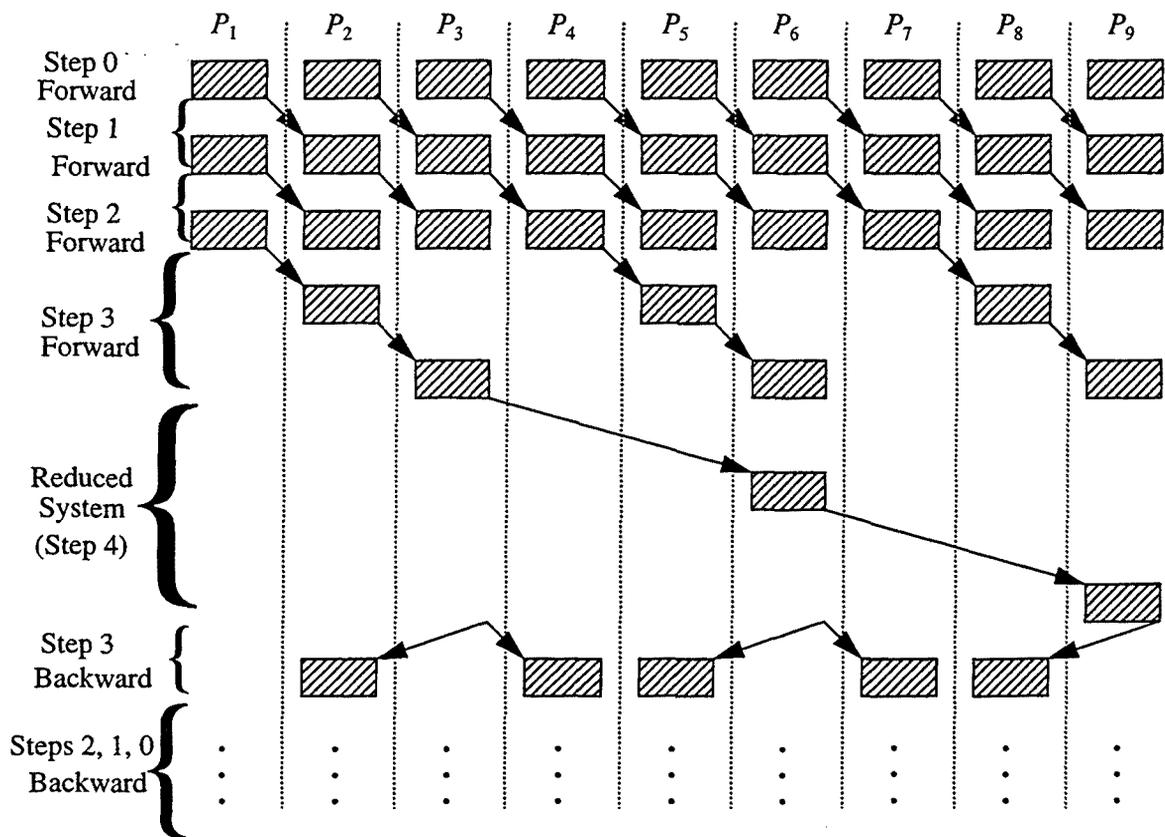


Figure 5.8. Data transport and computation graph of R -CR with $N/P \gg R$.

The early termination of the two level R -CR algorithm implies that the reduced system (step 2) is not solved but steps 0 and 1 have to be performed. Note in this case that if only step 0 has to be performed, the two step R -CR reduces to the early terminated DC. Now we discuss our reasons for discarding the early termination of the two step R -CR. First of all, let us say that the number of equations per partition during step 0 for both DC and R -CR are the same given that we use the same number of processors for each method. Thus, if $R_{min} > N/P^{(0)}$, DC cannot be terminated early but also, step 1 of R -CR has to be computed. During step 1 of R -CR, a very small number of equations are assigned to each partition ($P^{(0)}/P^{(1)}$). This depends on the actual number of processors used for each step rather than on the size of the system. Thus, it is very improbable that

$$R_{min} = \left\lceil \frac{\log(\epsilon(1 - \delta^{-R}) / \|b^{(0)}\|_{\infty})}{\log \delta^{-R}} \right\rceil < \frac{P^{(0)}}{P^{(1)}}$$

given the small size of the partition $P^{(0)}/P^{(1)}$.

A model for the two step R -Cyclic Reduction

In order to find the models of the two step version of R -CR in which $N/P^{(0)} = R$ we use the time diagram of Figure 5.9 and the algorithm of Figure 5.9. The time diagram of Figure 5.9 shows an example of the solution of a certain problem with $P^{(0)} = 9$ processors during step 0; groups of 3 processors (1, 2, 3; 4, 5, 6 and 7, 8, 9) solving the partitions of step 1 in parallel and finally, $P^{(1)} = 3$ processors (3, 6, 9) solving the reduced system in cooperative form. This example is the same as that of Figure 5.7.

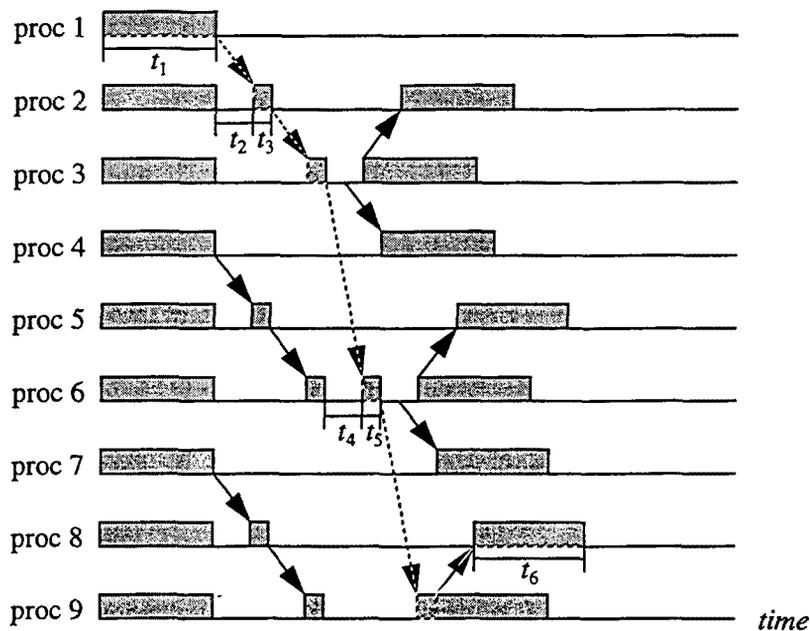


Figure 5.9. Time diagram and critical path of the execution of R -CR. t_1 is the execution time of the forward phase of step 0, t_2 is the communication time of 2 elements, t_3 is the computation time to solve one equation of a partition during step 1, t_4 is the communication time of 1 element, t_5 is the computation time to solve one equation of a partition during step 2 and, t_6 is the execution time of the backward phase of steps 1 and 0.

For the model, we suppose that $P^{(0)}$ processors are involved in step 0, $P^{(0)}/P^{(1)}$ processors are involved in the solution of step 1 and $P^{(1)}$ processors are involved in the solution of the reduced system. The execution time model determined by the time diagram is

$$t_1 + t_3 \cdot \left(\frac{P^{(0)}}{P^{(1)}} - 1 \right) + t_5 \cdot (P^{(1)} - 1) + t_6$$

for the computations and

$$t_2 \cdot \left(\frac{P^{(0)}}{P^{(1)}} - 1 \right) + t_4 \cdot \left((P^{(1)} - 1) + \left(\frac{P^{(0)}}{P^{(1)}} - 2 \right) \right)$$

for the communications.

In general, the model shown above is transformed into

$$\begin{aligned} & \left[13(N/P^{(0)} - 1) + 10 \left(\frac{P^{(0)}}{P^{(1)}} - 1 \right) + 6(P^{(1)} - 1) \right] T_S + \\ & + \left(2 \frac{P^{(0)}}{P^{(1)}} + P^{(1)} - 4 \right) \cdot T_{SU} + \left(3 \frac{P^{(0)}}{P^{(1)}} + P^{(1)} - 5 \right) \cdot T_E. \end{aligned}$$

The computation time modelled has a minimum in $P^{(0)} = \lceil K_1 N^{2/3} \rceil$ and $P^{(1)} = \lceil K_2 N^{1/3} \rceil$ where $K_1 = \lceil (K_5 K_3^{1/3}) / K_4^{4/3} \rceil^{2/3}$ and $K_2 = \lceil (K_5 K_3) / K_4^2 \rceil^{1/3}$. Also, $K_3 = 10T_S + 2T_{SU} + 3T_E$, $K_4 = 6T_S + T_{SU} + T_E$ and $K_5 = 13T_S$. If the optimum number of processors is larger than the available number, the optimum number of processors for step 1 turns to be $P^{(1)} = \sqrt{P^{(0)} K_3 / K_4}$ assuming that $P^{(0)}$ is equivalent to the number of processors available.

5.4.3 The Overlapped Partitions Method

As well as for the case of DC, the Overlapped Partitions Method suits parallel computers very well. Here, we suppose that all data are originally distributed as for the case of DC. Thus, given that each processor requires some equations of a neighbouring processor to form its overlapping, it is necessary to perform one communication at the beginning.

Figure 5.13 shows the algorithm that solves the partition corresponding to processor p . At the beginning of the parallel execution of OPM each processor sends the overlapping equations of its neighbouring processor. After that, Gaussian elimination is performed on the overlapping and non-overlapping equations.

```

if p > 1 then
    receive (am[1], ..., am[Rmin], taskid[p - 1])
    receive (bm[1], ..., bm[Rmin], taskid[p - 1])
endif
if p < P then
    send (a[R - Rmin + 1], ..., a[R], taskid[p + 1])
    send (b[R - Rmin + 1], ..., b[R], taskid[p + 1])
endif
if p > 1 then
    do r = 1, Rmin - 1
        bm[r+1] = bm[r+1] - bm[r] am[r+1]
    enddo
    b[1] = b[1] - bm[Rmin] a[1]
endif
do r = 1, R - 1
    b[r+1] = b[r+1] - b[r] a[r+1]
enddo

```

Figure 5.10. Parallel implementation of the Overlapped Partitions Methods.

A model for OPM

Figure 5.11 shows a time diagram of OPM for the algorithm of Figure 5.10. With the time diagram of Figure 5.11, the model is as follows. The critical path is formed by communications to send the overlapping equations t_1 and the amount of work for the overlapping t_2 and the central part of the partitions t_3 . The communications can be performed in parallel and we suppose that both vectors a and b can be send within one send primitive. For this reason, we suppose that one start-up is performed and $2R_{min}$ elements are sent in each communication. The amount of time for this is $T_{SU} + 2R_{min}T_E$. The amount of computations for processors with a logic identification p larger than 1 are equivalent to $4R_{min} + 5R$. The reason for this is that for the overlapping we perform 2 loads per equation plus 2 arithmetic operations and no stores because we do not need but the result to equation R_{min} of the overlapping. For the rest of equations we perform 2 loads per equation plus 2 arithmetic operations plus 1 store. So, the total time to execute OPM is

$$(5.3) \quad (4R_{min} + 5N/P) T_S + T_{SU} + 2R_{min}T_E.$$

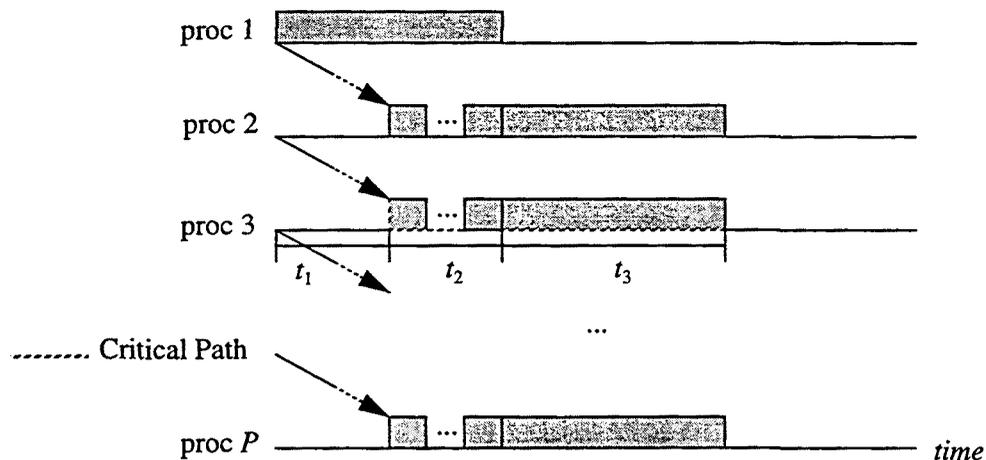


Figure 5.11. Time diagram and critical path of the execution of OPM. t_1 is the time to send the overlapping equations to the neighbouring processors, t_2 is the execution time to process the overlapping, t_3 is the execution time to process the central part of the partitions. Times t_1 and t_2 vary depending on δ and ϵ .

5.5 Comparison of the methods on different types of architectures

In this section, we compare the execution of the algorithms using the time models studied above for different communication times. The three different architectures analyzed can be summarized as shown in Table 5.1. There, we take the time to execute one scalar instruction $T_S = 1.0 \cdot 10^{-7}$ as a basis to compare the models of the algorithms varying the values of T_{SU} and T_E .

With the three combinations of values we measure the effect of very cheap communications (architecture type 1), the effect of very expensive start-ups (architecture type 2) and the effect of expensive communication times per element for expensive start-ups (architecture type 3). With those three types of architectures we cover the possible practical types of architectures. Also, varying one of the parameters from one type of architecture to another gives a measure of the influence of that parameter on the overall execution time of the algorithm.

	type 1	type 2	type 3
T_{SU}	$5 \cdot T_S$	$500 \cdot T_S$	$500 \cdot T_S$
T_E	T_S	T_S	$25 \cdot T_S$

Table 5.1. Types of architectures analyzed measured by the time to start-up a communication and send an element relative to the time to perform a computation.

For the comparisons performed in this section we suppose that we have 128 processors available. In case that the optimum number of processors given by the formulae obtained above are larger than 128 we suppose that only 128 processors are used.

For the analysis of the algorithms we compare the execution time, the number of optimum processors used and the efficiency obtained by the algorithms. The efficiency of an algorithm executed on P processors is equal to

$$(5.4) \quad E_P = \frac{T_1}{P \cdot T_P}$$

where T_1 is the execution time of the fastest sequential algorithm to solve the problem (Gaussian elimination in this case) and T_P is the execution time of the parallel algorithm on P processors.

With the different plots shown for each type of architecture we measure the absolute speed of the algorithms and the amount of resources used as well as the percentage of exploitation of those resources by each algorithm. Finally, for each architecture we give plots with the absolute suitability of the different algorithms (both non-early and early terminated) given δ and N for two values of the error allowed ϵ ($\epsilon = 10^{-7}$ and $\epsilon = 10^{-16}$) as we did for vector processors.

We start by analyzing architecture type 1. In this case we suppose that the start up of a communication is very small $T_{SU} = 5 \cdot T_S$ and the time to send an element is small, too $T_E = T_S$. The plots of Figure 5.12 show, for each method, the modelled execution time, the number of processors used and the efficiency for $100 \leq N \leq 10000$. Figure 5.13 shows the absolute comparison for $\epsilon = 10^{-7}$ and $10 \leq N \leq 2500$.

Plot 5.12.a shows the execution time of the algorithms. There, we can see that R -CR behaves better than DC for non-strictly diagonal dominant systems. Also, it is important to note that Gaussian elimination is slower than both R -CR and DC. Note that for R -CR we need more processors than for DC (fig. 5.12.c), in particular, for $N > 1200$ we use 128 processors while for DC we are always below that number. Also, the efficiency is larger for R -CR than for DC for $N > 3000$ (fig. 5.12.b).

When we turn to the early termination of DC and OPM, we can see that for large values of R_{min} or k (weak diagonal dominance and small maximum error allowed), OPM is significantly faster than the early terminated DC. In this case, OPM is the fastest algorithm only for large values of N because the two step R -CR is faster for small values of N . Note that although R -CR is faster for small values of N , the efficiency of OPM and the early terminated DC (fig. 5.12.b) is larger because they use a smaller number of processors than R -CR (fig. 5.12.c). For small values of R_{min} or k (large diagonal dominance and maximum error allowed) OPM and the early terminated DC behave similarly and are clearly faster than the other methods. Also, their efficiency is much larger for $N > 2000$ because the size of the partitions can be made small.

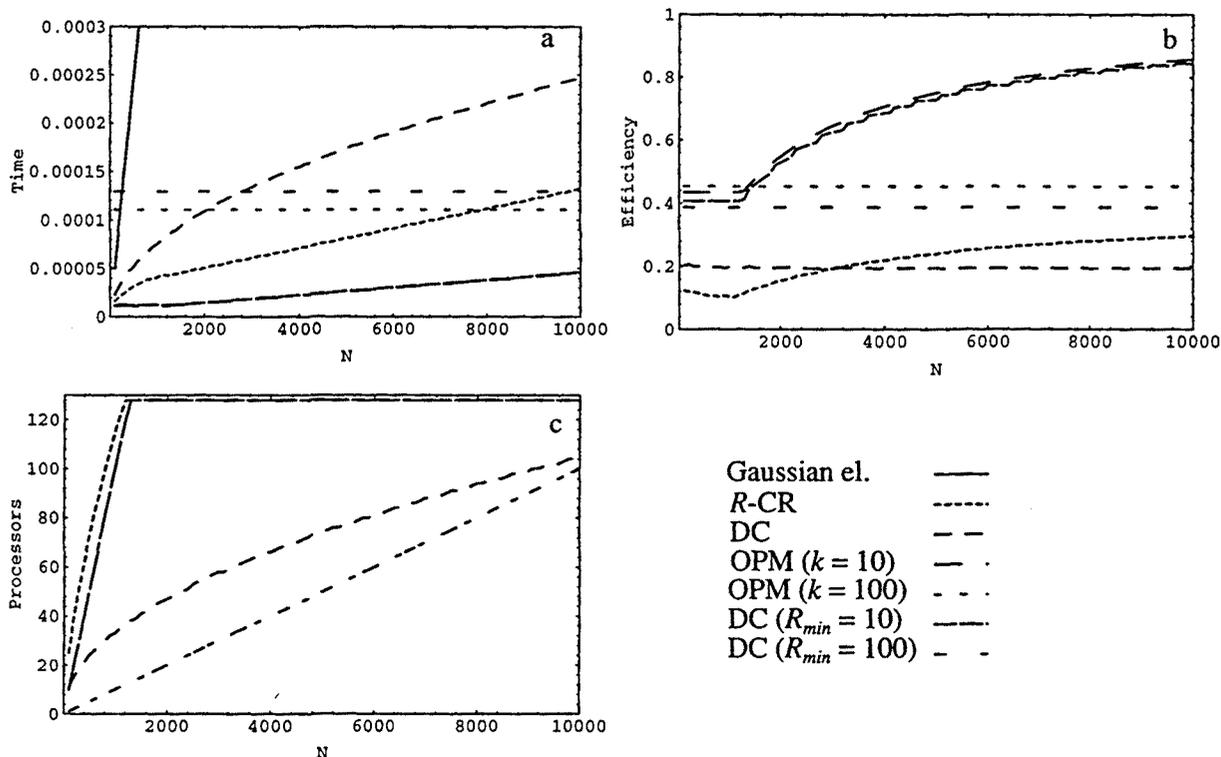


Figure 5.12. Comparison of the models for architecture type 1. a) Execution time, b) Efficiency and, c) Optimum number of processors.

The absolute comparison of the methods for $\epsilon = 10^{-7}$ (fig.5.13) shows that R -CR is faster than the rest of the methods for weak diagonal dominances and, in general, for small systems. On the other hand, OPM and the early terminated DC are faster than the other methods for large diagonal dominances and matrix sizes. Note that there is an area where all the methods are approximately similar. Also, there is a triangular area where OPM is faster than the other methods. Here, we do not show the plot for the absolute behaviour of the methods for $\epsilon = 10^{-16}$ because R -CR is always the fastest method for the cases plotted.

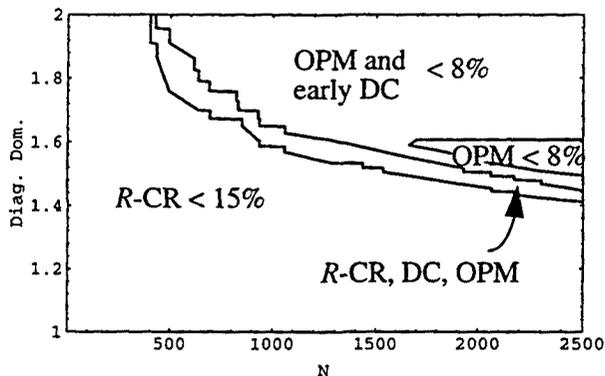


Figure 5.13. Absolute comparison of the algorithms on architecture type 1 for a maximum error allowed $\epsilon = 10^{-7}$.

With architecture type 1 we measure the behaviour of the algorithms with almost no communication time. The far end of this comparison would be the measure of the execution time with costless communication. In this case the optimum number of processors would be the number of available processors P (128 in our case) unless $N < P$. With this we would get the maximum efficiency attainable.

Now we turn to architecture type 2. Figure 5.14 shows plots of the execution time, the optimum number of processors and the efficiency of the algorithms on a supposed computer with a very costly start-up time of the communications $T_{SU} = 500 \cdot T_S$. In this case, we keep the time to communicate an element equal to the case of architecture type 1 in order to measure the influence of very large start-up times. Also, Figure 5.15 shows the absolute comparison of the methods for $\epsilon = 10^{-7}$ and $\epsilon = 10^{-16}$.

The execution time in this case is mostly dominated by the weighty communication start-ups. This is specially reflected in the non-early terminated algorithms where R -CR and DC have more than 80% of their execution time devoted to communication. It is important to recall that R -CR and DC perform a series of sequential communications that are equivalent to the number of processors, so, the larger the number of processors the larger the execution time. For this reason, Gaussian elimination is faster than the parallel algorithms for small systems of equations. On the contrary, the early terminated algorithms perform significantly better than the non-early terminated algorithms because they can perform their communications in parallel. In this case the total number of communications are equal to the number of processors but equivalent to one communication in terms of time consumed.

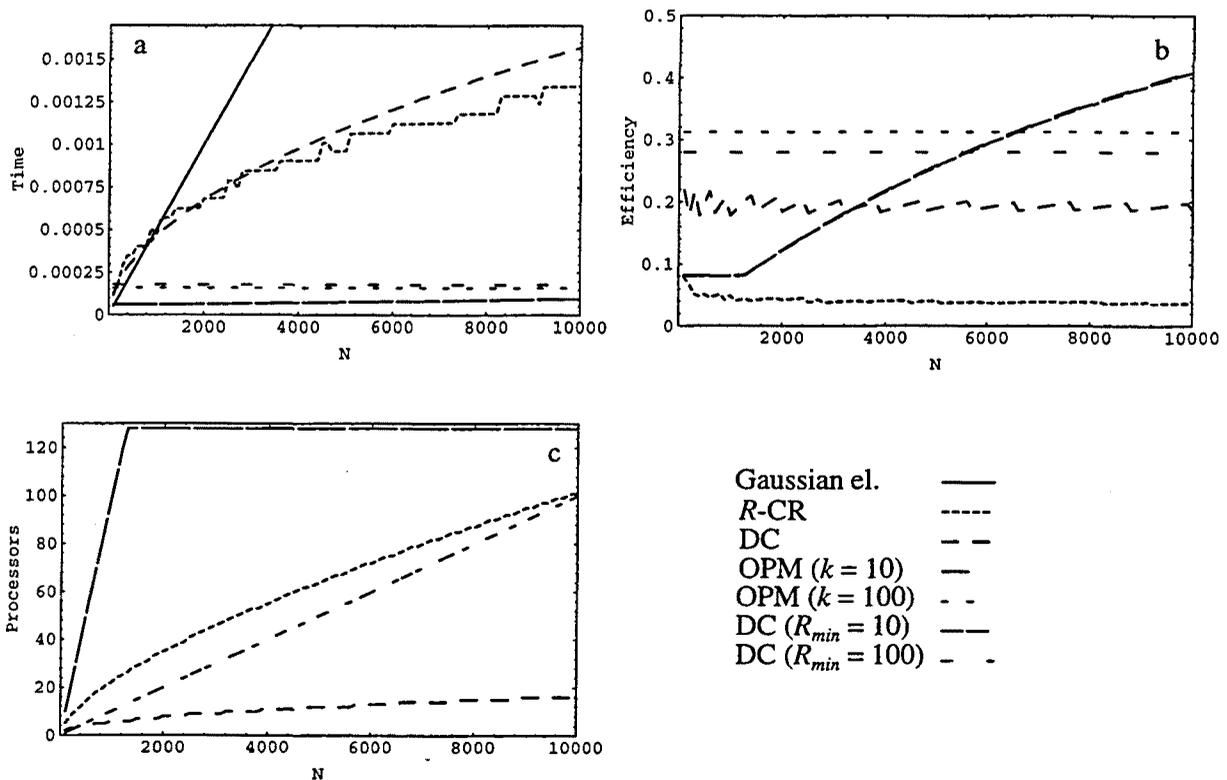


Figure 5.14. Comparison of the models for architecture type 2. a) Execution time, b) Efficiency and, c) Optimum number of processors.

The number of processors needed to achieve minimum execution times changes significantly from type 1 architecture. Here, DC requires a small number of processors, i.e. less than 20 in all the cases while for architecture type 1 DC needs a bit more than 110 for $N = 10000$. This is a natural tendency of the formulae that determine the optimum P to reduce the number of communications when those are significantly expensive. The case of R -CR is very similar although the number of processors needed in this case is larger than for DC (far more than 4 times). Note here that the execution times of DC and R -CR are similar so, the efficiency of DC is larger. The number of processors that determines the minimum execution time for the early terminated algorithms is the same as for the case of type 1 architecture. The reason for this is that this minimum execution time is independent of the number of processors.

Note that the efficiency of the algorithms is considerably smaller than for architecture type 1. The obvious reason for this is that most of the time is spent in communicating a small amount of data which at the end reflects a very poor exploitation of the resources.

The absolute comparison of the algorithms on architecture type 2 reflects what we said during the previous paragraphs. On one side, OPM and the early terminated DC are the fastest algorithms for medium to large systems and medium to large diagonal dominances. Nevertheless, in the case of small systems and weak diagonal dominances, Gaussian elimination is the fastest algorithm. DC and R -CR are the fastest algorithms in some cases of weak diagonal dominance and medium to large systems of equations.

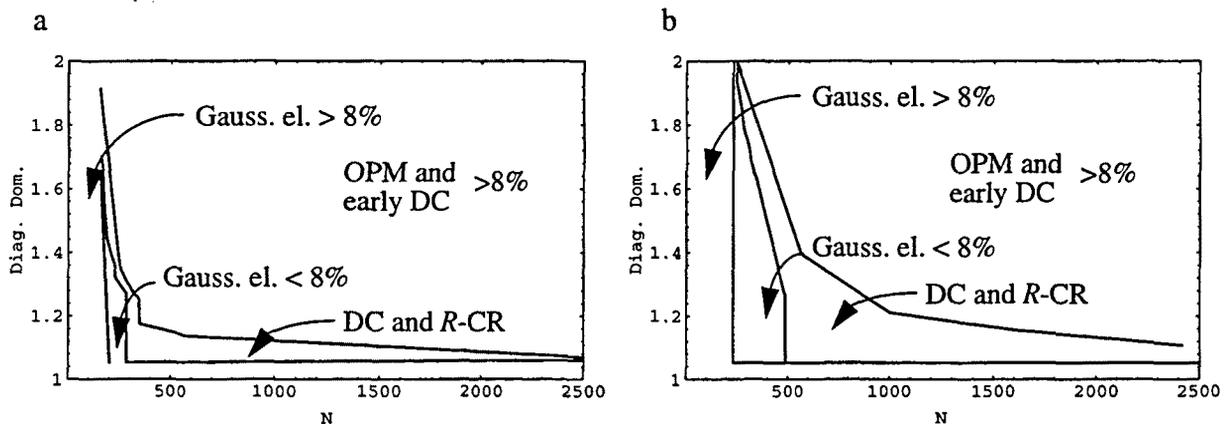


Figure 5.15. Absolute comparison for the algorithms on architecture type 2. a) Maximum error allowed $\epsilon = 10^{-7}$, b) maximum error allowed $\epsilon = 10^{-16}$.

Finally, we analyze architecture type 3. Figure 5.16 shows plots of the execution time, the optimum number of processors and the efficiency of the algorithms on a supposed computer in which the communication time per element has been incremented to $T_E = 25 \cdot T_S$ while $T_{SU} = 500 \cdot T_S$. With this we measure the influence of the communication time per element on the algorithms studied. Figure 5.17 shows the absolute comparison of the methods for $\epsilon = 10^{-7}$ and $\epsilon = 10^{-16}$.

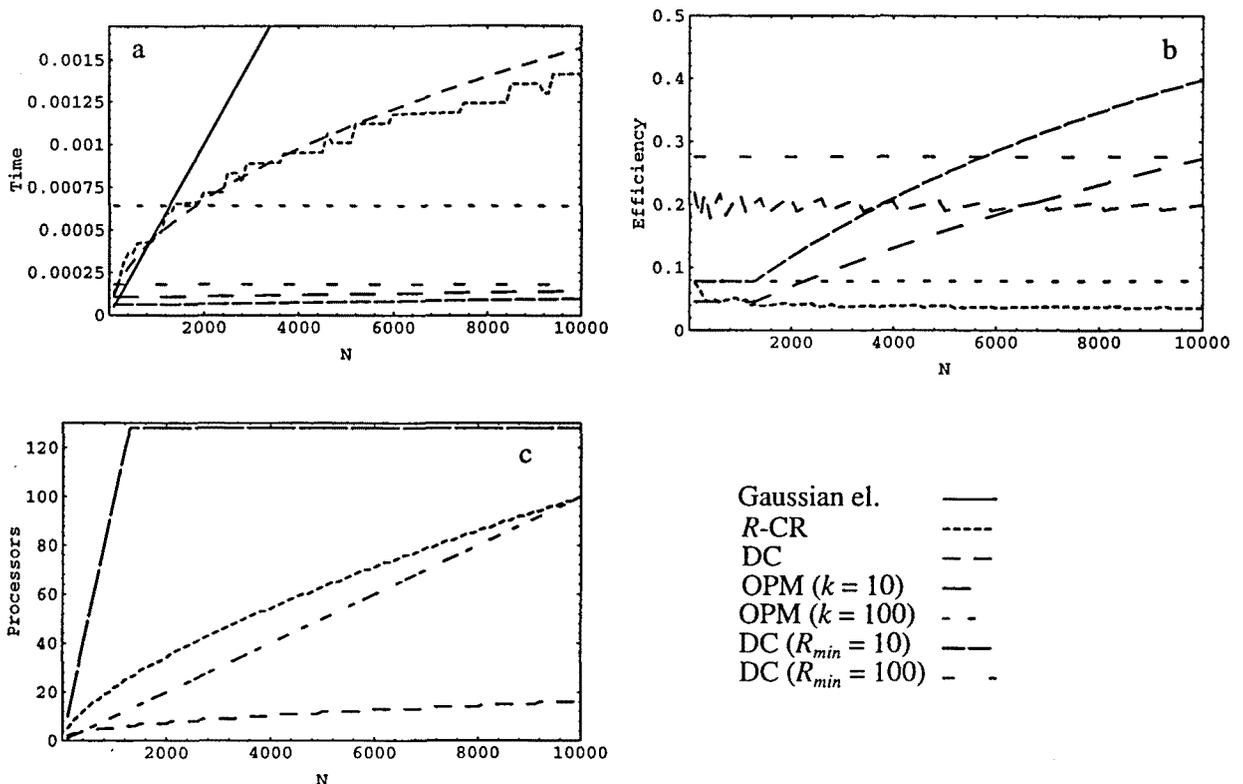


Figure 5.16. Comparison of the models for architecture type 3. a) Execution time, b) Efficiency and, c) Optimum number of processors.

In this case, the execution time of the algorithms is mainly dominated by the communication time as in the case of architecture type 2. The main difference now is that OPM turns to be the slowest of the methods for strictly diagonal dominant systems of equations. The reason for this is that at the beginning of the execution of the algorithm all the overlapping data are exchanged between neighbour processors. This implies that we are sending a considerable amount of elements per message for OPM. The other methods do not change noticeably because their messages involve only one or two elements which changes the execution time very little.

Note that the efficiency decreases for OPM with respect to architecture type 2 because its execution time has increased while the number of processors that are used is the same.

In terms of the absolute comparison of the algorithms, we can say that apart from slight differences, the plots in Figure 5.17 for architecture type 3 are very similar to the plots of Figure 5.15 for architecture type 2. The main difference in this case is the fact that OPM is not competitive with the early terminated DC for large systems and diagonal dominances.

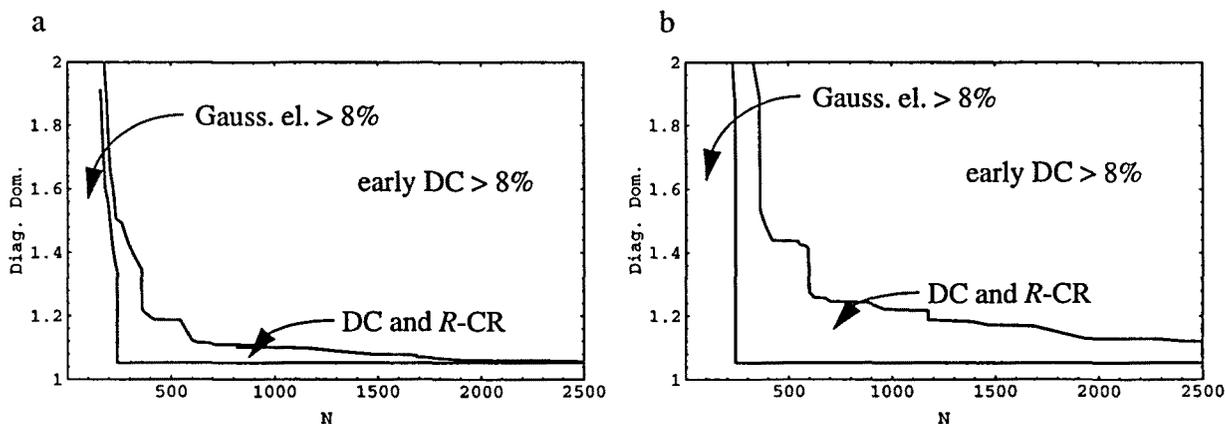


Figure 5.17. Absolute comparison for the algorithms on architecture type 3. a) Maximum error allowed $\epsilon = 10^{-7}$, b) maximum error allowed $\epsilon = 10^{-16}$.

An example on the Pyramid 16/860SYS

In this section we show an example of the execution of Divide and Conquer on the Pyramid 16/860SYS. Also, we compare the real execution time to the execution time models built above.

The reason for not using the Pyramid to validate the models built here is that it does not adapt exactly to the model architecture described above. Each processing node of the Pyramid is more complex and includes a superscalar processor that cannot be modelled with the simple assumptions made above. Also, the communication has more restrictions than those assumed above. As a consequence, although the fit of the execution time model for DC gives accurate results this is not true for the case of other algorithms on the Pyramid. Nevertheless, the general tendency of the algorithms on this specific architecture is very similar to that of architecture type 3 as we have noticed after implementing some of the algorithms.

The Pyramid 16/860SYS

In this section we use the Pyramid 16/860SYS by Transtech and the communication primitives of PVM [Tran94,Tran94a] to implement the version of DC explained above.

A scheme of the Pyramid parallel computer is shown in Figure 5.18. Each of the 16 processing nodes has a 50MHz INTEL i860 superscalar processor and a 16Mbyte memory. Each node is connected to a communication board with a Transputer T805 and 4Mbytes of memory. Each communication board has four communication channels of which two are connected to one crossbar to interconnect processors. One of the other two channels is connected to a crossbar to communicate with the host computer and the other one is connected to another crossbar as a channel to mass storage. The speed of these communication channels is 20Mbits/sec. In Figure 5.18 we show the connections explained here for one only processing board for simplicity. The host computer is a SUN workstation and has four channels to the crossbar with the Pyramid.

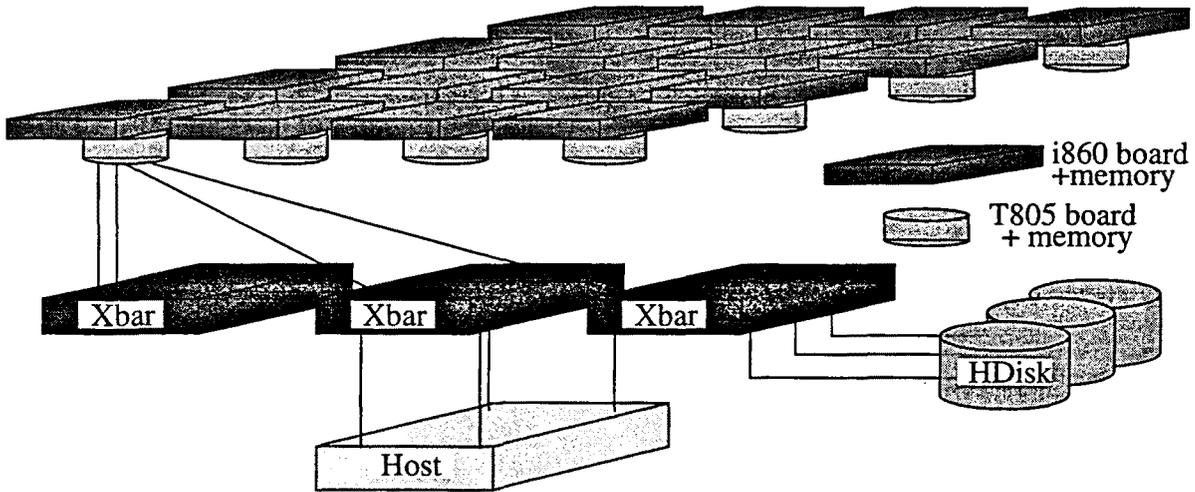


Figure 5.18. Structure of the Paramid 16/860SYS.

PVM on the Paramid 16/860SYS

PVM is a software that allows the use of a network of Unix computers as if they were a single large parallel computer [GBDJ94]. The popularity of PVM is given by the portability of the software designed with it. For this reason, many parallel computer manufacturers have introduced the primitives or subsets of them in the communication libraries of their computers. This is what happens with Transtech and the parallel computer Paramid 16/860SYS.

The Paramid version of PVM has some differences with the standard version of PVM described in [GBDJ94] as we comment below. The standard PVM allows the user to spawn a series of tasks onto a set of different processors. Also, it allows any set of two processors to communicate by means of send and receive routines assuming that there is a unique channel of communication to which all processors are connected. With this, it is not necessary to specify the way in which processors are connected at the cost of making simultaneous sets of communications more expensive. This programming advantage of PVM turns out to be a restriction of the Paramid version of PVM. The reason for this is that the Paramid requires the explicit hardware configuration to be specified at compile time because PVM is implemented using the Parmacs communication library. The PVM default hardware configuration of the Paramid is a linear array of processors in which only neighbouring processors can communicate directly. Other hardware configurations have to be programmed by the user making programs written in PVM not so easily portable as the PVM philosophy intends.

A remarkable difference between the computer model that we have described above and the use of PVM on the Paramid is that the latter does not allow a process to simultaneously be sending and receiving different messages. This is equal to suppose that PVM assumes one only channel per process. This makes the Paramid slightly different from the model of computer described in a previous section and used throughout the chapter.

Implementation of DC on the Paramid 16/860SYS

The implementation of DC requires that only neighbouring processors communicate. So, the default hardware configuration offered by the Paramid version of PVM can be used. Here we compare the real execution times given by DC on the Paramid 16/860SYS to the execution times given by the model of the algorithm built above. The model has been obtained by fitting different executions of DC to

$$T_{DC} = K_1 \cdot (N/P) + K_2 \cdot P + K_3$$

which is equation (5.1) to which the constants have been substituted by variables. The fit of the model for the execution of DC on the Paramid 16/860SYS gives the following constants

$$K_1 = 1.089 \cdot 10^{-6} \quad K_2 = 5.6 \cdot 10^{-4} \quad K_3 = 7.9 \cdot 10^{-4}$$

and the mean quadratic error of the fit is less than 5%. The optimum value of P for this model is

$$P = \lceil 4.4 \cdot 10^{-2} \cdot \sqrt{N} \rceil.$$

Figure 5.19 shows the execution time of DC on the Paramid 16/860SYS using 2, 4 and 8 processors and the execution time given by the execution time model explained above. Note in Figure 5.19 that 2 processors are faster than 4 and 8 processors on the Paramid in some cases. The optimum P predicts that 2 processors work faster than 4 or 8 processors for small values of N .

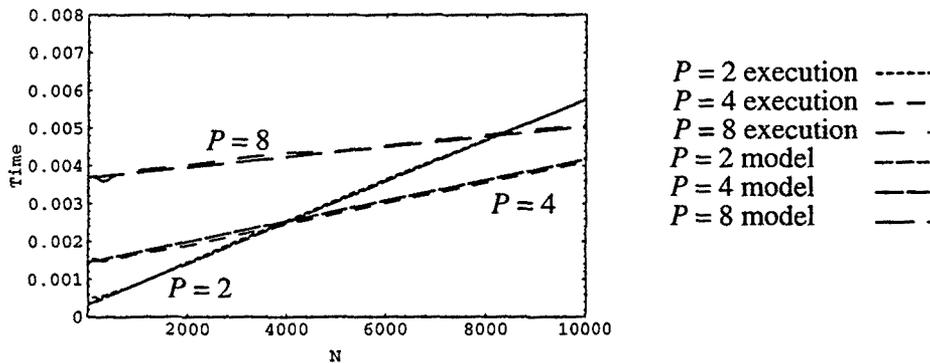


Figure 5.19. Comparison of the algorithms and the models of DC.