The Recursive Decoupling Method for Solving Tridiagonal Linear Systems

by

GIULIA SPALETTA, Dott.

A Master's Thesis

Submitted in partial fulfilment of the requirements

for the award of Master of Philosophy

of the Loughborough University of Technology

September, 1991

Supervisor: Professor D. J. EVANS, D.Sc.

Declaration

I declare that this thesis is a record of research work carried out by me, and that is my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

G. SPALETTA

Acknowledgements

I wish to express my deepest and most sincere gratitude to Professor D. J. Evans for giving me the opportunity to carry out this work, in the first instance; and subsequently for his friendly and unfailing guidance, continuous help and inspiring enthusiasm throughout this research. Finally, for his invaluable advice and infinite patience during the writing of this thesis. Thanks to Professor Evans, I can consider the period I spent studying under his supervision as one of the most fruitful experiences both in my academic career and life.

I also wish to thank:

- Mrs. J. Poulton for her professional help and constant, friendly presence;
- Mr. M. Sofroniou for his interest and fruitful collaboration, for many stimulating discussions, not to mention his typing of the whole of chapter 5 (core of this thesis) and the improvements he has brought to the original manuscript. Most of all, I wish to express to him my special indebtedness for being such a patient, true friend;
- Miss. H. Y. Sanossian, Mr. N. M. Bahoshy and Dr. A. Osbaldestin for their active co-operation and constant support as colleagues, but most of all as very good friends;
- Dr. W. S. Yousif and Mr. G. S. Samra for all their technical advice (and their infinite patience);
- Miss. L. Howard for her help in typing part of this thesis;
- all my colleagues and the staff of the Department of Computer Studies.

Finally, I thank my family for their love and understanding.

Abstract

Abstract

The work presented in this thesis mainly concerns the analysis of parallel algorithms for the solution of tridiagonal linear systems and the design of a new tridiagonal equation solver, which can be run on a MIMD (Multiple Instruction Multiple Data stream) type parallel computer, in particular the Balance 8000 Sequent system at Loughborough University of Technology.

In the first chapter, an introduction to the existing computer models is given, together with a brief description of the process that has led from the uniprocessor machine to the development of different parallel architectures. Enhancement is given to MIMD shared memory systems. In this respect, the main characteristics of the Sequent system are presented, as well as the main programming features supported by the Balance Operating System, the Dynix.

The second chapter presents the fundamentals of parallel programming on the Balance 8000 computer. Terms and concepts that are specific to multitasking programs are introduced. Also, the two multitasking methods, data partitioning and function partitioning, are outlined. In the same chapter, we investigate problems (such as program dependencies, sharing of data, synchronization of concurrent process) arising from the adaptation of an application to parallel versions, and the related programming techniques. Some of the parallel programming tools are described, with particular attention to the so-called "data partitioning with Sequent Fortran" and "data partitioning with Dynix".

Chapter 3 starts with an outline of the most well known algorithms for the solution of tridiagonal systems, one of which is analysed in more detail in chapter 4. Parameters used to evaluate performance are defined, such as

speed-up, efficiency and computational complexity, together with the basic principles of Parallel Numerical Analysis.

In the fourth chapter, the Wang tridiagonal system solver is presented. We have considered a variant of this partitioning method suitable for MIMD architectures, and we have modified it to run on the Balance 8000. Test matrices have then been used, in order to evaluate the performance of the Wang routine on the Balance computer and to form a comparison with the new Recursive Decoupling routine of chapter 5.

The fifth chapter constitutes the core of the whole thesis. The new algorithm also belongs to the class of partitioning methods, since it is based on repeated partitioning of the coefficient matrix into 2x2 submatrices; this strategy, together with a rank-one updating procedure, allows us to calculate the solution explicitly, by solving independent sets of subsystems. Furthermore, the methods turns out to be intrinsically parallel and suitable for solution on multiprocessor architectures. The performance of the Recursive Decoupling routine on the Balance 8000 computer has been tested by using the same example matrices as those used to test the Wang method.

The thesis concludes with a chapter summarizing the main results and suggestions for further research.

Keywords

Tridiagonal Linear Equations; Shared Memory Parallel Computers; Sequent Balance 8000 Multiprocessor; Partition Method; Recursive Decoupling Method; Parallel Numerical Analysis.

Contents

Contents

1. Introduction to Parallel Computers

1.1. Introduction

In the last few years we have seen an explosion in the interest in parallel processors and parallel programming.

The scope of parallel processing is to reduce the elapsed time to complete a job.

This time will basically depend on the coding style, the architecture of the machine and the hardware implementation.

The job of everybody in charge of software development (system designers, compiler and library writers, programmers) is to get the actual time required by the calculations as close as possible to the ideal.

Tools have been developed to express the parallelism explicitly, either in the form of subroutine libraries or language extensions; furthermore, studies are still in progress, concerning the automatic parallelization of sequential code.

To date, the only automatic system available is limited to individual loops. Parallelism at a higher level must still be specified by the programmer.

1.2. A Classification of Computer Models

A knowledge of the computer architecture and the hardware implementation is not essential to the programmer. However, when performance becomes critical, a good understanding of the hardware parallelism can be fundamental to the program's tuning.

In spite of all the efforts made to write portable programs, some algorithms will run efficiently on certain architectures, poorly on others.

The situation is worse for parallel processors than for uniprocessors, due to the wider variety of architectures.

We can state a classification of different computer models, based on those aspects in the hardware implementation of parallelism that most affect the coding style [16]:

1) shared memory systems (figure1.1);

2) distributed memory systems, also called message passing systems (figure1.2 & figure 1.4);

3) hybrid systems (figure1.3).

We are mostly interested in the first type of computer architecture, therefore we shall present a brief study of this kind of parallel machine.

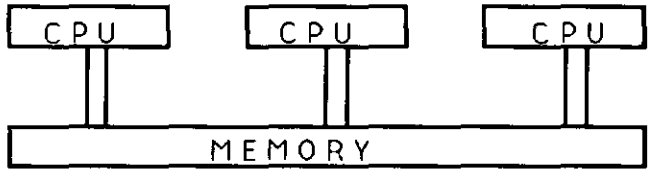


FIGURE 1.1. Schematic of a shared memory system.

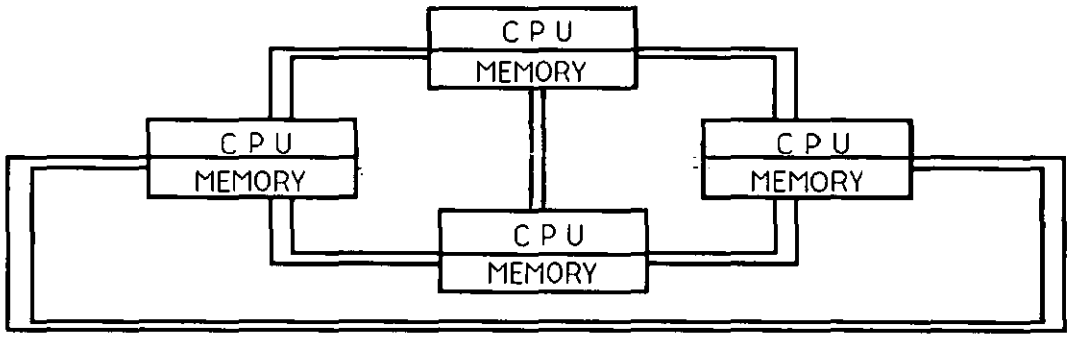


FIGURE 1.2. Schematic of a distributed memory system: fully interconnected message passing machine.

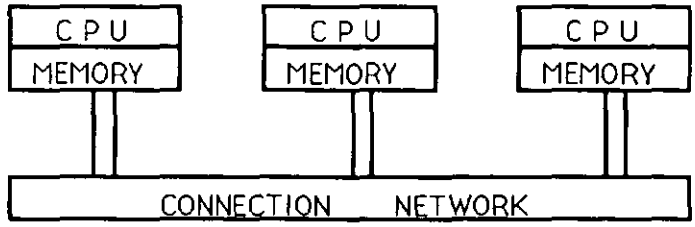


FIGURE 1.3. Schematic of a hybrid machine.

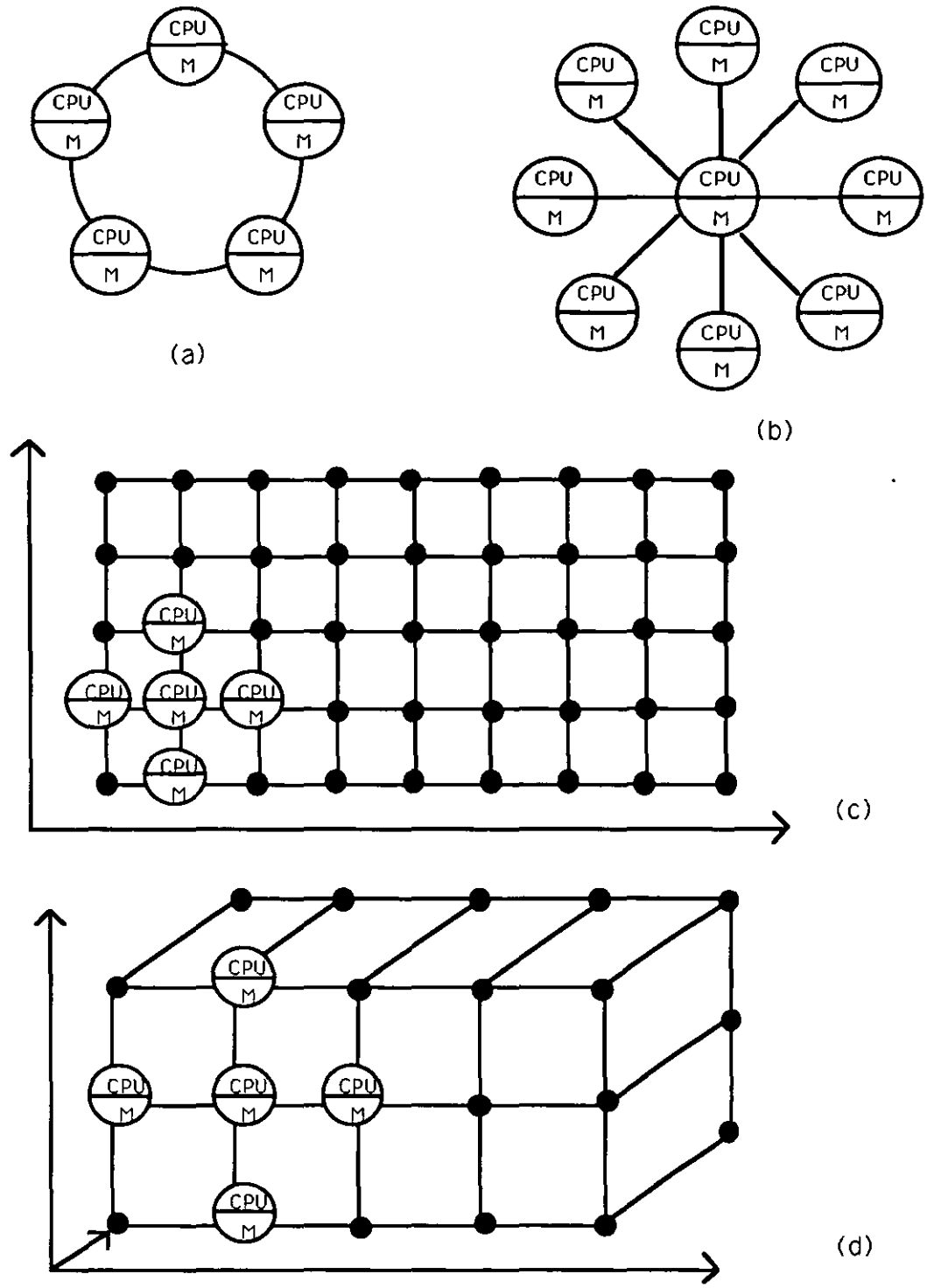


FIGURE 1.4. Distributed memory systems. (a) Ring connection machine. (b) Star connection machine. (c) Mesh machine. (d) Hypercube of order 3. (M: memory).

1.3. Shared Memory Systems

A shared memory machine has a single global memory accessible to all processors.

Each processor may have some local memory (such as "registers" on the Cray X-MP or the "cache" on the IBM 3090).

The data organization inside the memory (global and local memory) is totally transparent to the user.

The data access time is independent of the processor making the request.

This is not to say that there is no memory contention. Problems like page faults, memory bank conflicts, etc., still affect the performance.

Algorithms are easy to design for shared memory systems.

The data input on these machines is done as if running on a uniprocessor.

On the other hand, programs are hard to debug.

The most common type of error involves picking up wrong data from a global variable.

There is no indication of when the error occurred, so that the computing process continues, producing an erroneous final result. Data organization, therefore, is a key to parallel algorithms, even on a shared memory computer.

Unfortunately, the most commonly used language for scientific purposes (Fortran) only allows quite simple data structures (just scalars and arrays), inducing the programmer to concentrate on program flow rather than on data management. The latest version of Fortran language permits the use of a wider variety of structures and mechanisms. The data sharing specifications, though, still constitutes a fundamental problem on shared memory systems, a problem that becomes even more critical when the parallelism is nested.

To simplify the programmer's job, in this last case, most parallel processors provide only a single level of parallelism; that is to say that a master process is allowed to spawn subprocesses, while the subprocesses may not themselves spawn processes.

Data is either known to all the created processes or is private.

As a consequence of everything that has been said so far, the shared memory systems need a few language extensions.

Firstly, the need to declare which data is private to each processor (local data) and which is known to all processors (global data) arises.

Secondly, synchronization is needed to prevent out-of-sequence access of different processors to the shared memory.

The following considerations answer the above mentioned problems.

The work in a shared memory machine is usually divided up in a so-called "fork-join" style: one process spawns the subprocesses (fork) and waits for them to finish (join).

A means to restrict access to the code is needed and obtained, introducing the concept of a "critical section"; this is a section of code executed by all processors, one at a time (such as in the case of a *reduction variable*).

The concept of a "sequential section" is also introduced, which is a part of the code that has to be executed by only one processor and skipped by all the others. A sequential section is typically used to initialise global data .

The easiest way of obtaining synchronization is the JOIN construct. When this is not possible, other constructs have to be used, such as "barriers" or

"semaphores". All these concepts will be more precisely illustrated in the following paragraphs.

Finally, since the cost of sharing data is very small in shared memory machines, programmers often tend to parallelize the code at the Do-loop level. In the case of independent loop iterations, each processor can run a different subset of the loop index range, providing that each index value is used exactly once.

There are basically two ways of parallelizing a Do-loop. One way is to assign the first loop index value to the first arrived processor, the second index value to the second processor, and so on. Whenever a processor has completed its task (its loop iteration), it returns to the top to get more work. In this way, an automatic load balancing is realized. On the other hand, this way of obtaining a parallel Do-loop requires some form of synchronization, to assure that each processor gets a unique value of the loop index.

A second way to parallelize a Do-loop is to partition it so that each processor will do a certain set of loop iterations. This way of proceeding is to be preferred if the work is naturally load balanced, and expecially if the synchronization cost is high.

1.4. Parallel Numerical Analysis and the Flynn Classification of Computer Models

In classical numerical analysis, a universal computer model is represented by the Von Neumann machine; this can be schematized as follows (figure 1.5):

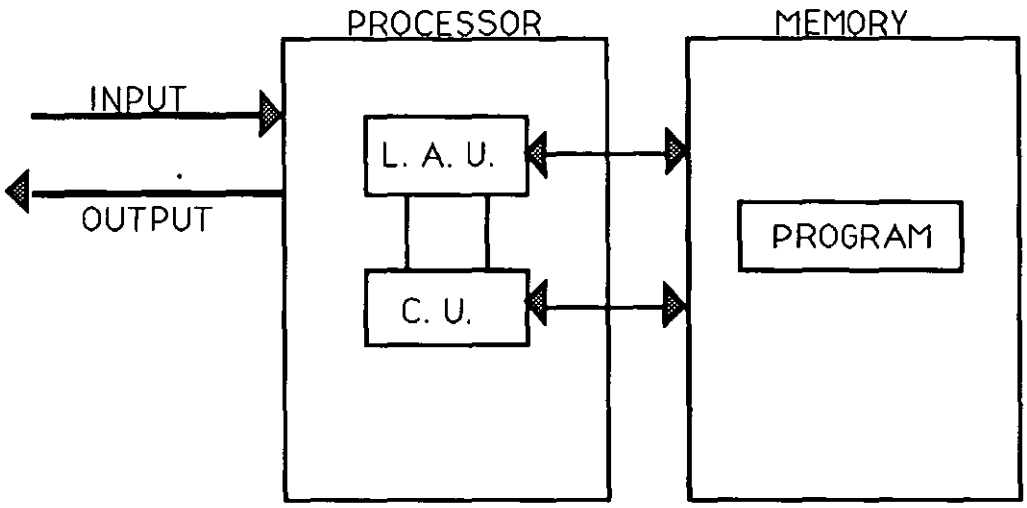


FIGURE 1.5. Scheme of the Von Neumann machine.
L. A. U. Logic & Arithmetic Unit.
C. U. Control Unit.

The main features of this universal computer are:

a) digital representation of variables;

b) serial processing, carried out according to the basic operations of arithmetic and logic;

c) the program is a coded version of the algorithm to be implemented;

d) data are held in the main memory.

The algorithms of classical numerical analysis are then based on the Von Neumann model and entail a large number of elementary operations.

This basic serial model has been taken as the starting point for all further developments, until the concept of "parallelism" began to be discussed.

Parallelism was to be interpreted in the widest sense, that is not just to build a parallel digital computer, but also to create a body of numerical mathematics

which exploits the possibilities offered by parallel computers. Furthermore, the question arose as to whether there exists a maximal parallelism for a given range of problems.

All these facts led to the need for a "parallel numerical analysis". Connected to this need was the problem of formulating a standard machine model for parallel numerical methods.

During the last thirty years, the performance of serial machines has been improved greatly, due to the use of a new technology and new design.

Parallel features have been introduced:

- in the organization of input/output channels;
- by overlapping the execution of instructions;
- by using interleaved storage techniques.

Starting from these ones, new developments have been realized, leading to a truly parallel machine. Gains have been obtained, such as:

1) increase of computing speed;
2) possibility of solving problems too complex for serial computers;
3) exploitation of the inherent parallelism of some problems;
4) possibility of calculation of a solution in real time.

On the other hand, parallel computers present new difficulties, due to a complicated organization of the data and also due to machine dependent optimization for efficiency.

At present, there is still no standard model for parallel systems. Such a model could be represented as shown in the following figure:

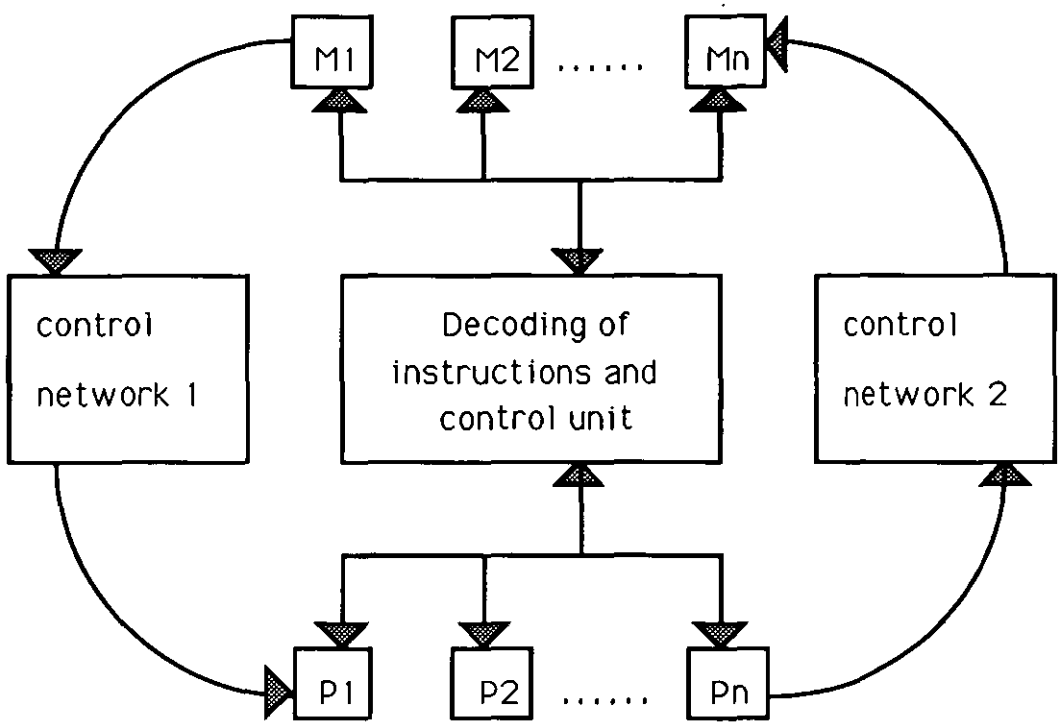


FIGURE 1.6. General configuration of a parallel computer with different levels of parallelism (M: memory; P: processor).

In the above diagram parallelism is possible at different levels:

- within the control unit;
- among processors;
- among the stores;
- in the data network.

The above figure, though, is too general both for the building of a functioning computer and the development of algorithms. Such a standard diagram can only be taken as a theoretical basis for parallel numerical analysis and parallel computers.

Depending on which level of parallelism is implemented in the diagram of figure 1.6, we can state the following classification of computer (this classification is due to Flynn [13]):

1) SISD machines: it is the Von Neumann model (Single Instruction - Single Data stream);

2) SIMD machines: array processors, pipeline processors and associative machines belong to this class (Single Instruction - Multiple Data stream);

3) MIMD machines: computers with several data processors and multiple processor systems belong to this class (Multiple Instruction - Multiple Data stream);

4) MISD machines: it has been proven that this type of organization (Multiple Instruction - Single Data stream) is equivalent to that of a Von Neumann machine. Therefore the MISD class is considered empty.

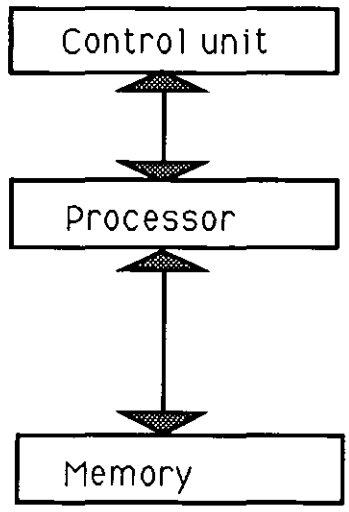


FIGURE 1.7.
Scheme of a
SISD computer.

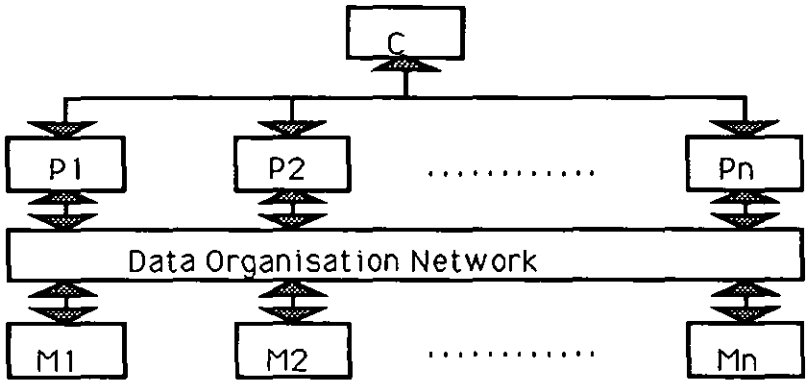


FIGURE 1.8. Scheme of a SIMD computer.

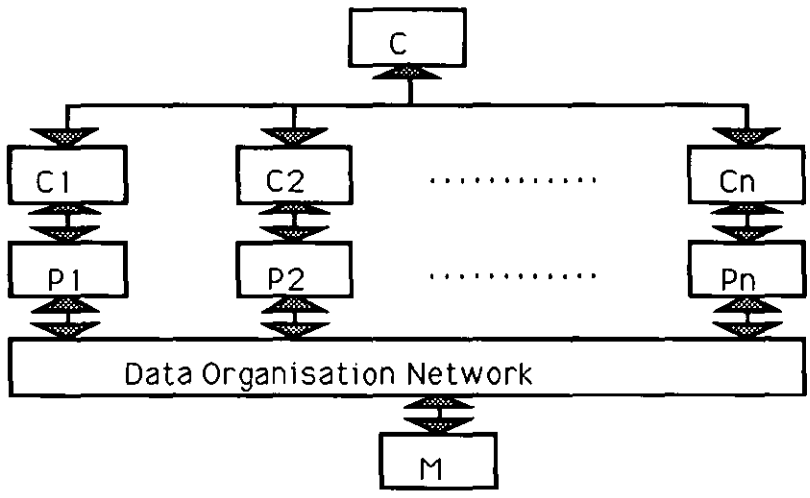


FIGURE 1.9. Scheme of a MISD computer.

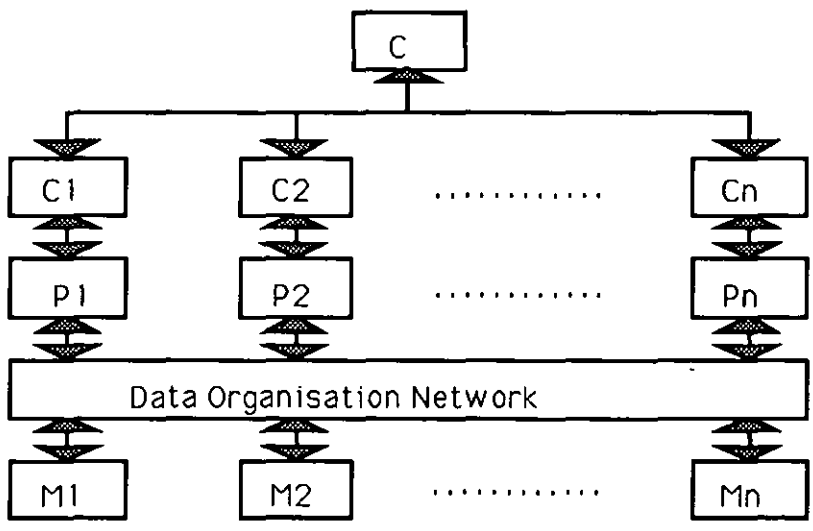


FIGURE 1.10. Scheme of a MIMD computer.

NOTE. C: control unit; P: processor; M: memory.

In the context of parallel numerical analysis, all these computer models involve problems of rounding errors and their propagation, together with questions of numerical stability of the algorithm used.

The SIMD organization, in particular, is suitable for classes of numerical problems such as:

- matrix operations;
- numerical integration of differential equations;
- MonteCarlo methods;
- pattern recognitions.

MIMD machines consist of a certain number m of independent processors P1, P2, .., Pm, each having its own control unit (C1, C2, .., Cm respectively). All these processors share, among other things, a number of input/output units and a main memory.

At every instant each processor can carry out different instructions in parallel, that is to say all processors can operate simultaneously. Unlike the SIMD machines, the MIMD computers are considered as "general purpose" computers, because they are much more flexible than the SIMD ones and a greater variety of problems can be solved through them.

As mentioned before, in this work we are only concerned with true multiprocessor shared memory machines; an example of this kind of machine is represented by the Balance 8000 computer.

In the following paragraph we will briefly introduce the Balance architecture and the parallel programming capabilities of this system.

1.5. The Balance 8000 Parallel Processing System

The Balance 8000 Sequent system is a multiprocessor shared memory machine and therefore it belongs to the MIMD class. Its main features are the following ones [24]:

a) it is a true multiprocessor, consisting of multiple identical processors (CPUs); each CPU is a general purpose 32 bit microprocessor;

b) it is a shared memory machine, i.e. there is a single common memory; an application can consist of multiple processes, all accessing shared data held in the memory;

c) it is a tightly coupled machine, i.e. all processors share a single pool of memory; sharing memory is a natural way for two processes (running on different processors) to communicate with each other.

   Note that a tightly coupled multiprocessor can do more than assign non-interacting processes to a different processor. It can also distribute a single process among many processors, so that each processor only executes part of the calculation. This is done, as we will see in the following chapter, to get a "speed-up" (that is if a process takes time t to run on an uniprocessor, it could take time t/n to run on n processors);

d) the Balance system has a symmetric architecture, since all processors are identical and can execute both user code and operating system code;

e) there is a single high-speed Common Bus, used by all the processors, the memory modules and the input/output controllers: this is done to simplify the adding of processors, memory and input/output bandwidth;

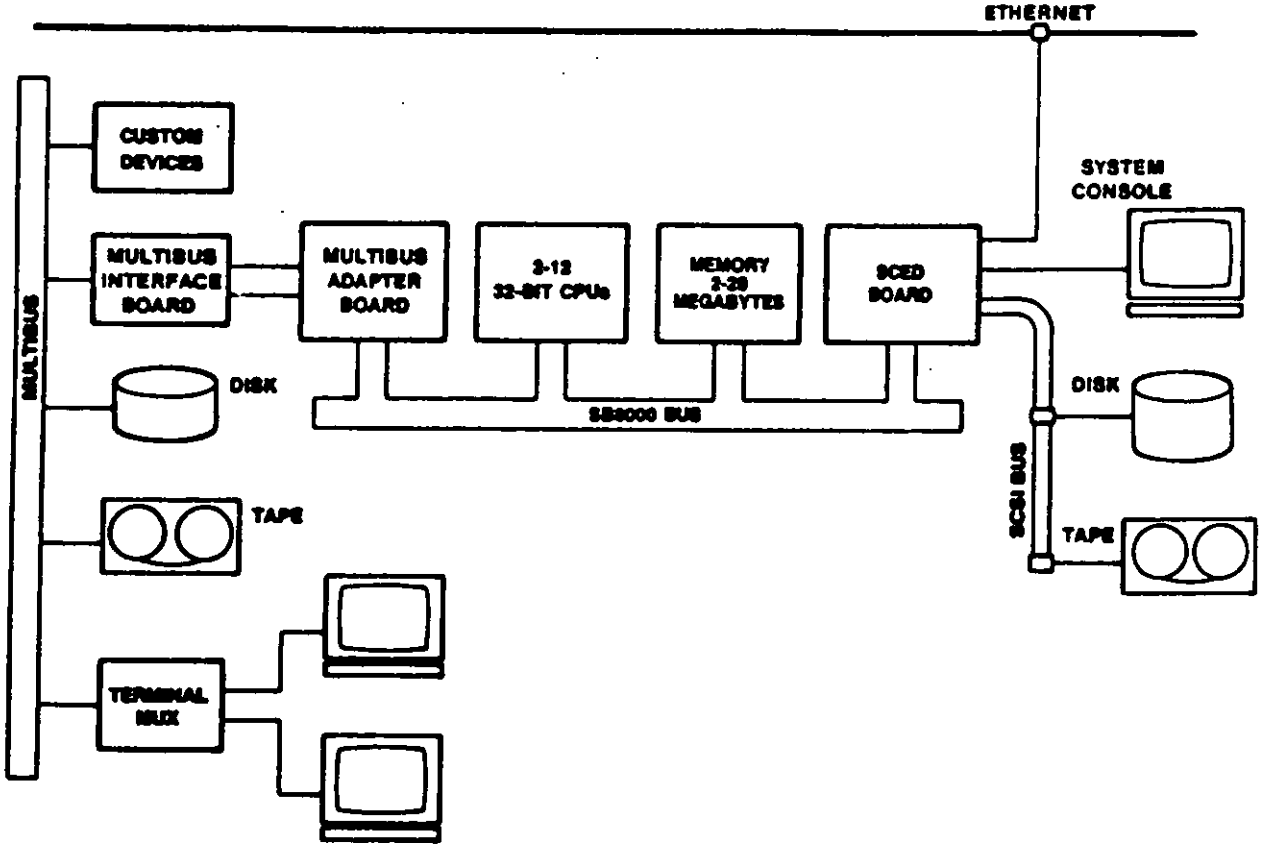
f) programs written for a uniprocessor system can run on the Balance system in such a way that it appears transparent to the user; that is programs do

not need to be modified for multiprocessing support. Processors can be added or removed, with no need of modifying either the operating system applications or the user applications;

g) dynamic load balancing is provided automatically by the processors, to ensure that all processors are kept busy (in the most efficient possible way) as long as there are executable processes available;

h) hardware support for mutual exclusion is provided, to enable the user to lock any section of physical memory, whenever there is the need for exclusive access to shared data structures.

The following figure illustrate the components of a typical Balance 8000 system (taken from Sequent Computer System, *"Balance 8000 System Technical Summary"*,[26]):



Processors

The Balance 8000 computer is designed to employ from two to twelve 32 bit CPUs, in a tightly coupled multiprocessing architecture.

The CPUs are packaged two per board.

To change the number of CPUs in the system it is necessary only to shut down the system and add or remove one or more dual-processor boards. No changes to the operating system or user applications are required.

Memory

The Balance 8000 can employ from 2 to 28 Megabytes of primary memory and it can provide 16 Megabytes of virtual address space per process.

Memory is packaged in one-board or two-board memory modules.

Memory can be added or removed in much the same way as the CPUs.

SCSI bus

The SCSI bus (Small Computer System Interface bus) is used to connect block-oriented devices, such as disk drivers or tape drivers to the system. It supports high-speed, high-volume data transfer between memory and peripherals (disks, tape units).

SCED board

A Balance 8000 system can include from 1 to 4 SCED boards (SCSI Ethernet Diagnostic controller boards). Each SCED board can serve as host adaptor on a SCSI bus.

In any Balance 8000 system one SCED board is designated the "master" SCED board: this master board connects to the system console and provides

power-up diagnostics. It also provides a power-up monitor for any program running on the main CPU, such as programs to boot the operating system.

Multibus interface

A Balance 8000 system can include up to 4 Multibus interfaces: they enable the system to incorporate any of a variety of peripherals and custom devices.

The Balance 8000 System bus

It is a high-performance data bus, tailored to multiprocessing in the sense that it provides the high bus bandwidth needed to support multiple CPUs.

The Balance System bus is a 64 bit system bus which carries data among the CPUs, the memory modules and the peripheral subsystems.

Network interfaces

A Balance 8000 can connect to up to 4 other systems both in local area networks (one per SCED board), using Ethernet, and in wide-area networks, using ordinary telephone lines.

The connection in local area networks facilitates communication among users as well as the sharing of files and devices.

Each of the four connectable Ethernet local area networks can connect hundreds of systems, over distances of one mile or more.

Furthermore, the Balance system networking capabilities include those common to all modern Unix systems.

Terminal multiplexor

This is a two-board module that resides on the Multibus and can connect to a terminal, printer, modem or other compatible device.

There can be up to 4 terminal multiplexors per multibus.

Operating system: the Dynix

The Dynix operating system is a version of Unix 4.2BSD modified to exploit the Balance parallel architecture; differences between Dynix and Unix 4.2BSD are transparent to the user.

Dynix also supports most utilities, libraries and system calls provided by Unix System V and, like other versions of Unix, it is a multi-user operating system. Two or more users can use the system simultaneously, while each user seems to have the system's undivided attention.

This is achieved through an operating system technique called *multiprogramming*: a CPU moves from one process to another many times per second, so that the computer system is allowed to execute multiple unrelated processes (programs) concurrently. All the executable processes wait in a "run queue": when the CPU suspends or terminates the execution of one process, it switches to the process at the head of the run queue.

The Dynix operating system uses the same technique, except that multiprogramming on Dynix is enhanced by the Balance multiprocessing architecture: in a Balance system a pool of processors is available to execute processes from the run queue. Dynix balances the system load among the available processors, keeping all processors busy as long as there is enough work available.

Note that the Dynix operating system does multiprogramming for all the users automatically.

Along with the multiprogramming technique, the Balance system also supports another kind of parallel programming: *multitasking*.

Multitasking is a programming technique that allows a single application to consist of multiple closely co-operating processes [9].

As a consequence of multitasking and multiprogramming, we can make the following considerations.

By definition, parallel programs execute concurrently, meaning that at any instant the system is executing multiple programs. On a Balance system, parallel programs execute simultaneously: at any instant, the Dynix operating system can be executing multiple instructions from multiple processes (one process per CPU).

Thus, parallel programming on a Balance system has two special benefits:

- multiprogramming yields improved "system throughput" for multiple unrelated programs. That is, each program finishes in about the time it would take on a uniprocessor (which is running that program alone);
- multitasking yields improved "execution speed" for individual programs, that is the owner of an application (consisting of multiple processes) sees an improvement in the execution speed of the application itself, beyond what would be possible on a uniprocessor.

In the following chapter we will analyze parallel programming on the Balance 8000, using the multitasking technique.

2. Principles of Parallel Programming on the Balance 8000

2.1. Introduction to Parallel Programming on the Balance 8000

As illustrated in section 1.5, the two basic kinds of parallel programming are multitasking and multiprogramming. This chapter is primarily about multitasking, since the Dynix operating system of the Balance 8000 does multiprogramming for all users automatically.

Many applications can be converted from sequential algorithms to parallel algorithms with relative ease, yielding linear or quasi-linear performance improvements, as more CPUs are dedicated to the task.

In addition, certain types of applications can be designed specifically to exploit the Balance multiprocessing architecture.

The gain in the execution speed, that can be achieved by means of the multitasking technique, is determined by the following factors:

- the percentage of the program's time that can be spent executing parallel code (a great number of applications need to spend less than 2-3% of their time executing sequential code);
- the number of processors available to the application;
- the hardware contention imposed by multiple processors competing for the same resources (such as the system bus, the system common memory, etc.). Note that on a Balance system the overhead due to this hardware contention is negligible, since most CPU memory operations access cache memory, not the system bus;
- the overhead in creating multiple processes;
- the overhead in synchronization and communication among multiple processes.

In adapting an application for multitasking, therefore, we will aim to run as much of the program in parallel as possible; at the same time, we will aim to balance the computational load as evenly as possible among parallel processes.

2.2. Parallelizable Applications: Homogeneous and Heterogeneous Multitasking

We also have to determine whether an application can benefit from parallelization and which kind of multitasking technique is the most suitable.

A parallel application, in fact, consists of two or more processes executing simultaneously. These processes can be multiple instances of the same program ("homogeneous multitasking" or "data partitioning") or they may be distinct but co-operating programs ("heterogeneous multitasking" or "function partitioning").

Homogeneous multitasking basically consists of running the same code on each CPU.

Multiple identical processes are created and work on different portions of the data structure simultaneously.

Data partitioning, therefore, applies to applications performing many iterations on large data structures (e.g. matrix multiplications, Fourier transformations).

The entire data structure can be divided up evenly among processes, before they start work (static load balancing), or each process can work on one portion at a time, going back for more work when it finishes (dynamic load balancing).

Heterogeneous multitasking, on the contrary, assigns different code to each CPU; that is, all the processes work simultaneously on a shared data set but each process handles a different task.

Applications performing many different operations on the same data set are candidates for function partitioning (e.g. flight simulation, program compilation).

While some applications require function partitioning or a combination of data and function partitioning, most problems adapt more easily to data partitioning.

This last method offers some advantages over function partitioning, such as less programming effort is required to convert a serial program to a parallel algorithm. Furthermore, with data partitioning it is easier to achieve an even load balancing among processors; it is also easier to adapt the programs automatically to the number of available processors.

In the remaining part of this chapter, we will only refer to the homogeneous multitasking technique.

As far as it concerns the decision whether to parallelize a program, we can point out that many programs spend the majority of their time executing in very few routines (usually just one or two). When converting a program to a parallel version, it is often possible to achieve maximum gain in execution speed simply by parallelizing these few routines. Furthermore, a typical fraction of code that cannot be parallelized turns out to be just 2-3% for most programs (as already been mentioned).

Typical sections of code that have to be performed serially are those related to initialisation phases and input/output operations.

2.3. Program Dependencies

Once the portions of parallel code have been identified, the next step is to analyse all the possible program dependencies, for any program unit [2, 3, 24].

Some program operations, in fact, may depend on previous operations, while some may be executed in any order. *Program Dependence Analysis*, therefore, is needed to carry out all the ordering necessary to guarantee correct results.

When a program unit has no dependencies, the statements in that unit can be executed in any order or even simultaneously.

Most of the time, this is not the case; we can group the kinds of dependencies into two classes: *data dependencies* and *control dependencies*.

Within the data dependencies class, we separate:

- flow dependence;
- antidependence;
- output dependence.

Flow dependence occurs when one operation sets a data value that is used by a subsequent operation:

I) $A = B + C$

II) $D = 3 \times A$

Statement (II) depends on the result of statement (I).

Antidependence occurs when one operation uses a memory location that is loaded by a subsequent operation:

I) $A = B + C$

II) $C = 3 \times B$

Statement (I) must execute before statement (II), since the first statement uses the current value of the variable C.

Output dependence occurs when one operation loads a memory location which is also loaded in a subsequent operation:

I) $A = B + C$

II) $A = D - 3$

Statement (II) must execute after statement (I), or A will contain the wrong value at the end of this program unit.

The second class of program dependencies is the *control dependencies class*; it includes dependencies due to the required flow of control in a program:

I) IF $(X.GT.0)$

II) $A = B + 3$

Statement (II) is conditionally executed, depending on the result of the test in statement (I).

It is necessary to identify all the program dependencies within a program unit (and for all program units), in order to transform a given program, loop or subroutine, to correctly run in parallel. It is also necessary to correctly organise the data structure (shared or private) and to get synchronization points and locking mechanisms for all the processes.

2.4. Elements of Parallel Programming

The remaining section introduces some elements of parallel programming that are not common in sequential programming.

We have already discussed the multitasking technique and the program dependence analysis.

What we still need to consider is:

- the creation of shared and private data;
- the creation and termination of multiple processes;
- the division of computing tasks among parallel processes ("scheduling");
- the synchronization of parallel processes;
- the mutual exclusion of parallel processes (locks mechanisms).

Let us study all these subjects, one at a time, in the above order.

Shared memory and shared data

The Dynix operating system allows any number of processes to share a common region of system memory.

Any process that has access to a shared-memory region can read or write in that region, in the same way it reads and writes in ordinary memory.

Shared memory provides a direct and efficient method for co-operating processes to share data. It also simplifies the conversion of sequential algorithms to parallel (and it simplifies this conversion much more than message-passing mechanisms or network-based machines).

Multitasking programs include both shared and private data. Shared data is accessible by all the processes, while private data is accessible by only one process.

The following figure 2.1 illustrates the virtual memory contents of a process (16 Megabytes of virtual memory are allocated for each process):

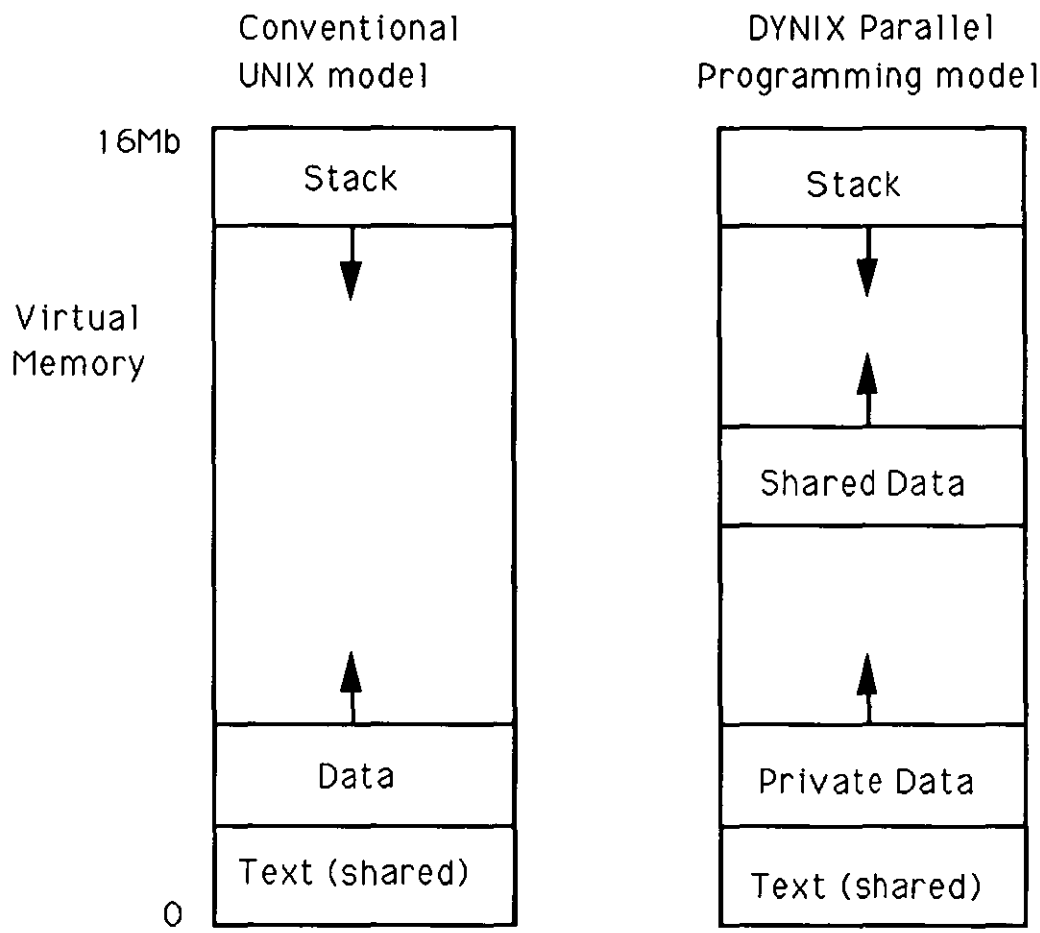


FIGURE 2.1. Comparison of virtual memory contents.

If the process forks any child processes (as we will see later), each child process inherits access to the parent's shared memory area and shared stack. Both the parent and the child processes can then access the shared data.

This mechanism (besides providing an efficient way of interprocess communication) uses less memory than having multiple copies of shared data; it also avoids the overhead of making such copies of shared data.

Process creation, scheduling and termination

In Dynix, as in other Unix-based operating systems, a new process is created by using a system call called a FORK. The new process *(child)* is a duplicate of the old process *(parent)*: the child process shares the same files and shared memory accessible to the parent process.

A process identification number *(process id)* distinguishes the parent process from all the created child processes: when some child processes are forked, the process id number 0 (zero) is assigned to the parent, while the process id number 1 is assigned to the first child process, the process id number 2 is assigned to the second child process, and so on. From this point on (until reaching the JOIN phase), they are separate entities.

The fork operation is relatively expensive. Therefore, a parallel application should fork as many processes as it is likely to need at the beginning of the program and terminate them at the end of the program (on completion of the program itself). If a process is not needed during certain code sequences, the process can wait in a busy loop (spinning) or it can relinquish its processors to other applications (until it is needed again).

In multitasking programming, tasks can be scheduled among all the processes created using three different techniques:

- prescheduling;
- static scheduling;
- dynamic scheduling.

In *prescheduling* the task division is determined by the programmer before the program is compiled. The programmer assigns a specific task to each process.

Automatic load balancing, therefore, is not allowed by the prescheduling technique (that only applies to heterogeneous multitasking).

In *static scheduling* the tasks are scheduled by the processes at run time, but they are divided in some predetermined (static) way.

The static scheduling procedure for one process is as follows:

1st step) it works out all the tasks that it will do;

2nd step) it does all its tasks;

3rd step) it waits until all other processes finish their work.

Static scheduling produces static load balancing: since the division of tasks is statically determined, some processors may stand idle while one processor completes its work.

This static technique only applies to homogeneous multitasking.

In *dynamic scheduling* the tasks are scheduled by the processes at run time and they are taken from a task queue.

The dynamic scheduling procedure for one process is:

1st step) it waits until there are some tasks to execute;

2nd step) it removes the first task from the task queue and executes it;

3rd step) if there are any more tasks to execute, it goes on to the second step. Otherwise, it goes back to the first step.

Dynamic scheduling produces dynamic load balancing: all the processes are kept busy as long as there is work to be done; the work-load is evenly distributed among the processes.

This dynamic technique applies to both homogeneous and heterogeneous multitasking.

Dynamic scheduling, though, entails more overhead than static scheduling: each time a process schedules a task for itself, it must check the shared task queue (to make sure that there is more work to do) and it must remove that task from the queue.

Process synchronization, loop scheduling and lock mechanisms

Synchronization is fundamental to ensure that each process performs its work without interfering with the other processes.

It is not unusual for a looping subprogram (to be executed in parallel) to contain a code section which depends on all the processes having completed execution of the preceding code.

All real application programs contain the program dependencies we have studied in section 2.3.

We then need some synchronization mechanisms to ensure the correct execution of multiple co-operating processes; these mechanisms are basically:

- barriers;
- locks and semaphores.

A *barrier* is a synchronization point: on reaching a barrier, one process marks itself as "present"; then it waits for all the other processes to arrive.

There are two kinds of barriers.

It is possible to synchronise all processes at a single pre-initialised barrier.

With the second type of barrier, the programmer is allowed to set more than one barrier or to synchronise just a subset of the processes.

A *lock* is the simplest kind of semaphore in the Balance Dynix system. It ensures that only one process at a time can access a shared data structure.

A lock has two values: locked or unlocked. Before attempting to access a shared data structure, a process waits until the lock associated with the data structure is unlocked (indicating that no other process is accessing the structure). The process then locks the lock, accesses the shared data and finally unlocks the lock.

While a process is waiting for a lock to become unlocked, it "spins" in a loop, producing no work.

It is impossible for two processes to acquire a lock at the same time. Even when a few processes attempt the same lock immediately, only one succeeds, while all the others have to wait (until the first process has released the lock).

Semaphores are synchronization mechanisms based on the locking/unlocking principle; they are used to protect order-dependent sections of code and to manage queues.

"Counting/queueing" semaphores, for example, are useful for queue management. When several processes are waiting for a lock, the lock will go to the first process that tries to acquire it right after it is unlocked. Counting/queueing semaphores can ensure that the lock is assigned (instead) to the process that has waited the longest for it.

If a barrier is used for synchronization, one process is delayed in a spinning state (called "busy-wait" state) until a set number of processes have reached the barrier.

When using a lock or a semaphore the situation is more complex; in this case, in fact, there exist four possibilities concerning what a process should do while it is waiting for its turn to access the locked code section. These four possibilities are:

1) the process does not wait; it performs a different task and checks the lock again later;

2) the process spins in a "busy-wait" state;

3) the process "blocks", that is to say it relinquishes its processors to another job;

4) the process spins for a specified period of time, then it blocks.

We complete this paragraph with a consideration affecting input/output handling.

Input/output, in parallel programming, is complicated by the need for caution when multiple processes write to the same file. These complications can usually be reduced by performing input/output only during sequential phases or by designating one process as a server to perform the input/output operations.

This chapter is concluded by introducing the parallel programming tools supported by the Balance system.

2.5. Parallel Programming Tools

The applications that can be adapted for parallel programming vary greatly in their requirements for data sharing, interprocess communications, synchronization, etc. [4]. To gain optimum speed-up, the programmer must develop an algorithm that meets these requirements (while still exploiting the application's inherent parallelism).

To aid in this effort, the Balance system supports programming tools that adapt to the needs of a wide range of applications.

We are mostly interested in two of these parallel programming tools:

- the Fortran Parallel Programming Directives (Sequent Fortran);
- the Parallel Programming Library (Dynix).

We illustrate these two tools in more detail in the following sections and we show how to employ them for data partitioning.

Fortran Parallel Programming Directives: data partitioning with Sequent Fortran

The Fortran Parallel Programming Directives support parallel execution of Fortran Do-loops. By interpreting these directives, the Sequent Fortran compiler can restructure a Do-loop for parallel execution.

The user prepares the program for the preprocessor by inserting a set of directives: these directives identify the loops to be executed in parallel; they also identify the shared and private data within each loop and any critical sections of the loop under consideration. Furthermore, the Fortran Parallel Programming Directives allow the user to control the scheduling of loop iterations among processes and the data division among all processes.

Ideally, the loop to be chosen for parallel execution should be an "independent" loop (i.e. a loop in which no iteration depends on the operations in any other iteration).

Otherwise, it is reasonable to choose a loop which accounts for a large portion of the computation.

Finally, in the case of nested loops, choose the outermost loop (if possible).

Once it has been determined which loop to prepare for parallel execution, it is necessary to analyse all the variables in that loop and to classify them into one of the following categories:

- shared variables;
- local variables;
- reduction variables;
- shared ordered variables;
- shared locked variables.

After this analysis phase, the user is ready to use the Fortran Parallel Programming Directives, to prepare the loop under consideration for parallel execution; these directives are listed in the following table (Table 2.2):

TABLE 2.2

DIRECTIVES	DESCRIPTIONS
C$DOACROSS	Identify Do-loop for parallel execution
C$ORDER	Start loop section which contains a shared ordered variable
C$ENDORDER	End loop section which contains a shared ordered variable
C$	Add Fortran statement for conditional compilation
C$&	Continue parallel programming directive

Parallel Programming Directives.

At this point, the preprocessor handles all the low-level tasks of data partitioning. By interpreting the directives, the preprocessor produces a program that performs the following features:

- sets up shared data structures;
- creates a set of identical processes;
- schedules tasks among processes;
- handles mutual exclusion and process synchronization.

All this is done in a way that is totally transparent to the user.

For more detailed information about the loop variables classification and the use of the Parallel Programming Directives refer to [24].

If necessary, the user is allowed to call the Dynix Parallel Programming Library routines (see the next section), in order to preserve the correct data flow within the loop.

Parallel Programming Library: data partitioning with Dynix

The Sequent Parallel Programming Library is a collection of C routines which allow the programmer to perform parallel Fortran programs (as well as C and Pascal programs).

This library includes three sets of routines:

1) microtasking routines (microtasking library);
2) routines for general use with data partitioning programs (data partitioning library);
3) routines for memory allocation in data partitioning programs (memory allocation library).

By means of them, the user is able to:

- create sets of processes to execute subprograms in parallel;
- schedule tasks among processes;
- synchronise processes among tasks;
- allocate memory for shared data.

As a result, programs that use the Parallel Programming Library can be made to balance loads automatically among processors and to adjust the division of tasks at run time (basing the division on the number of available processors).

Data partitioning with Dynix consists of the creation of multiple independent processes to execute iteration loops in parallel. This is done as follows:

a) each loop to be executed in parallel is contained in a subroutine;

b) for each loop, the program calls a special function (m_fork), which forks a set of child processes and assigns a copy of the subroutine to each process;

c) each forked process executes some of the loop iterations (either static or dynamic scheduling can be used);

d) when necessary, the subroutine may contain calls to synchronization routines (m_sync, m_lock, m_unlock, etc.);

e) when all the loop iterations have been executed, control returns from the subroutine to the main program.

At this point, the program either terminates the parallel processes (by means of the m_kill_procs routine), or it suspends their execution until they are needed again (m_park_procs and m_rele_procs routines), or it leaves the parallel processes to spin in a busy-wait state and then uses them later.

A complete list of all the routines available in the microtasking library, in the data partitioning library and in the memory allocation library is given in the following three tables (Tables 2.3, 2.4, 2.5).

TABLE 2.3

ROUTINES	DESCRIPTIONS
m_fork	Execute a subprogram in parallel
m_get_myid	Return process identification number
m_get_numprocs	Return number of child processes
m_kill_procs	Terminate child processes
m_lock	Lock a lock
m_multi	End single-process code section
m_next	Increment global counter
m_park_procs	Suspend child process execution
m_rele_proces	Resume child process execution
m_set_procs	Set number of child processes
m_single	Begin single-process code section
m_sync	Check in at barrier
m_unlock	Unlock a lock

Parallel Programming Library Microtasking Routines.

Note: the microtasking library is designed "around" the m_fork routine; any other routine belonging to this library should only be used in combination with the m_fork routine.

TABLE 2.4

ROUTINES	DESCRIPTIONS
cpus_online	Return number of CPUs on-line
s_init_barrier	Initialise a barrier
S_INIT_BARRIER	*C* Macro
s_init_lock	Initialise a lock
S_INIT_LOCK	*C* macro
s_lock, s_clock	Lock a lock
S_LOCK, S_CLOCK	*C* macro
s_unlock	Unlock a lock
S_UNLOCK	*C* macro
s_wait_barrier	Wait at a barrier
S_WAIT_BARRIER	*C* macro

Parallel Programming Library Data Partitioning Routines.

Note: the data partitioning library includes a routine to determine the number of available processors; it also includes several synchronization routines and their analogous C preprocessor macros (these macros are faster than the normal function calls, but they can add to the code size).

TABLE 2.5

ROUTINES	DESCRIPTIONS
brk, sbrk	Change private data segment size
shbrk, shsbrk	Change shared data segment size
shfree	De-allocate shared data memory
shmalloc	Allocate shared data memory

Parallel Programming Library Memory Allocation Routines.

Note: the memory allocation library consists of routines that allow data partitioning programs to allocate or de-allocate shared memory; these routines also permit a change in the amount of shared and private memory assigned to a process.

For more detailed information concerning the Parallel Programming Library usage, refer to the Sequent Guide To Parallel Programming [24].

Data partitioning with Dynix, as well as data partitioning with Sequent Fortran, requires an analysis of all the variables concerned with the section of code (Do-loop) to be performed in parallel. It is necessary to identify:

- shared variables, i.e. "read-only" arrays and scalars or arrays whose elements are referenced by only one loop iteration;
- private variables, i.e. variables that are initialised in each loop iteration before their values are used;
- dependent variables (reduction variables, ordered variables, locked variables).

After the analysis phase, the users can structure their microtasking program, using the following:

- decide how many parallel processes have to be forked, by means of the m_set_procs parallel programming routine or by using a default value computed by the Parallel Programming Library (the number of processors that are currently on line can be obtained through a call to the cpus_online routine);
- call the m_fork routine to execute each looping subprogram in parallel;
- suspend or terminate parallel processes between calls to looping subprograms;
- terminate all parallel processes after the last looping subprogram has been executed.

By using the parallel programming tools, the user is allowed to perform either static or dynamic scheduling, to handle all the dependent variables, to synchronise all the co-operating processes, etc.; as already mentioned, the user is also allowed to allocate or de-allocate shared memory.

The final phases of program compiling, executing and debugging follows.

We only mention the Dynix parallel symbolic debugger PDBX: if the program produces incorrect results, it is possible to use this Dynix debugger, based on DBX(a debugger widely used on Unix systems).

We conclude by mentioning the Dynix GPROF profiler. This utility creates a program execution profile, that is to say a listing that shows which subprograms account for most of the program execution time. Since these subprograms are best to execute in parallel, the gprof option turns out to be a very useful tool.

3. Parallel Numerical Analysis: the Tridiagonal Linear Systems Problem

3.1. Introduction

The solution of tridiagonal systems of linear equations appears very frequently as the nucleus of many scientific computational problems.

These same problems are generally well suited for solution on both parallel and vector architectures; consequently, parallel algorithms have been designed for multiprocessor systems (MIMD) and pipeline vector computers (SIMD) [21, Evans 7, 10, 11, Stone 28, 29].

A tridiagonal system solution, however, involves recurrence relations that represent a severe difficulty for the implementation of efficient parallel code.

The aim of this work is to review and analyse the main techniques for solving tridiagonal linear equations on parallel computers, with particular attention to the Wang algorithm, and to present a new technique of solution, the Recursive Decoupling algorithm.

In sections 3.2 and 3.3, we briefly introduce some fundamental principles for the construction of parallel methods and the related techniques for solving the above-mentioned problems of recurrence relations.

In chapter 4, we present a more detailed analysis of the Wang algorithm for the solution of tridiagonal linear systems. This algorithm will constitute a term of comparison for the Recursive Decoupling method described in chapter 5.

3.2. Performance evaluation parameters: speed-up and computational complexity

Speed-up

In parallel numerical analysis, it is important to be able to estimate the speed gain expected from the operation of p processors working in parallel. This is done by introducing the so-called "speed-up" ratio

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

where n is the dimension of the problem,

$T_s(n)$ is the time required for serial program execution,

$T_p(n)$ is the time required for parallel version execution.

The maximum speed-up possible is always equal to the number of processes used. However, in practice, the speed-up is often less than this.

According to Stone [30], there are four possible forms for $S_p(n)$:

1) $S_p(n) = k * p$ in problems such as matrix calculations, finite difference/element discretisation, etc.

2) $S_p(n) = k * p/\log_2(p)$ in tridiagonal linear system solving, linear recurrence formulae, evaluation of polynomials, sorting problems, etc.

3) $S_p(n) = k * \log_2(p)$ in searching problems, etc.

4) $S_p(n) = k$ in some nonlinear recurrence relations, in compiler operation, etc.

The constant k is a machine dependent quantity such that $0 < k < 1$ and $k \cong 1$.

The speed-up parameter describes how efficiently one is using multiple processes: the closer $S_p(n)$ is to the number of processes used, the more efficient the parallel algorithm is and vice-versa.

Note that the discussion of speed-up is in terms of processes and not processors [2]. This is done under the assumption that the number of processors available is greater than or equal to the number of processes.

In the case that, during a program run, the number of processes (usually stored in the variable NPROC) is greater than the number of physical processes, the speed-up ratio obtained may be less than the ideal maximum speed-up inherent to the program.

At this point, we want to analyse the parallelization of a single loop (since, in practice, this is mostly the case) and the related speed-up.

For a single loop, the time required for parallel execution is given by the summation of the following addends:

- time to create processes;
- time to execute loop;
- other overheads;
- time to destroy the child processes.

The time required to generate child processes from a single parent process depends on the implementation of the FORK function. In Dynix, a fork

operation takes about 55 milliseconds (although this varies with the size of the process).

The "other overhead" involves synchronization and any sequential section needed to consolidate all the partial results obtained by the processes created. Note that this part of the calculation has no analogue in the single-process version; therefore it is "overhead".

In order to calculate the time to execute the loop, we can make use of the following table 3.1 and also consider that in Dynix one iteration through a Fortran loop takes about 4.0 microseconds.

TABLE 3.1

Fortran Language	Operand	
Operation	4 byte integer	8 byte real
Addition	2.8μsec	11.7μsec
Multiplication	10.3μsec	11.5μsec
Division	14.8μsec	13.7μsec

Execution times for standard arithmetic operations, using 32 bit integers and 64 bit reals, performed in Fortran parallel programs, using the Balance Parallel Programming Library.

It is useful to consider the parallelized loop "in isolation", that is ignoring the other overheads in the program. In this way, programmers can get an idea of how effectively they have parallelized the core of the program.

The speed-up calculated in this way is called "ideal speed-up", while the speed-up measured by considering all overheads is called "true speed-up".

The ideal speed-up assumes that the execution time of the parallel program is determined only by the number of operations performed in the parallelized core loop.

On the contrary, the true speed-up is measured directly from the execution times, rather than from counting the operations in the parallel loop.

Even if the core loop has been efficiently parallelized, it is normal for the true speed-up to be less than the ideal one, because of overheads.

However, in practical applications, it is the true speed-up that is of interest.

Note that the speed-up is a measure of how effectively processes are utilized, and not necessarily a measure of the actual speed of execution.

For example, consider the following loop with NPROC = 2:

Do 10 $i = 1 + id,\ 5,\ NPROC$

task

10 continue

join processes

The first process ($id = 0$) does iterations $i = 1, 3, 5$, while the second process ($id = 1$) does iterations $i = 2, 4$. If these processes are doing the same amount of work on each iteration, when the first process performs its task for $i = 5$, the second process will do nothing.

If T is the time required for a single iteration through the loop, then the parallelized program will require a time of $3T$ to execute the loop.

The sequential version will execute the loop in time $5T$, so that the ideal speed-up for this loop execution (ignoring any other overhead) is $5T/3T = 5/3$, less than the number of processes.

Now we consider the following loop:

```
do 10 i = 1 + id, 6, NPROC
        task
10 continue
join processes
```

again with NPROC = 2, then the parallel execution time is still $3T$, while the sequential execution time is $6T$. The ideal speed-up is now $6T/3T = 2$, equal to the number of processes.

Note, however, that the inefficiency due to the uneven distribution of work becomes less important as the size of the loop increases.

Normally, the larger is the dimension n of the problem, the more the speed-up approaches its maximum.

On the other hand, it is necessary to be aware of the fact that increasing the number of parallel executing processes may result in a relative performance degradation, particularly if the size of the problem is small.

Computational complexity

Another way of comparing two parallel algorithms is obtained by means of the computational complexity parameter [22, 23].

The computational complexity of an algorithm is defined as the total number of operations related to the algorithm itself.

In the case of vector/parallel algorithms, analogous definitions of vector/parallel computational complexity hold, involving vector/parallel operations respectively.

Algorithms designed for a vector architecture are often applicable to a parallel computer with a limited number of processors and vice-versa.

However, there are important differences between the two types of machines that usually make the computational complexity parameter not sufficient for a correct comparison of performances.

The timing considerations for a parallel computer are very different to those for a vector computer.

Also, the characteristics of each individual machine, such as the size of central memory available , the data accessibility, the instruction set, and so on, can greatly influence the portability of a particular algorithm.

In the case of vector architectures, for example, it is very important to consider the length of vectors involved in an operation: this length gives the so-called *grade of parallelism* of the vector operation.

If two vector algorithms A and B present the same vector computational complexity, with A having a grade of parallelism equal to n and B equal to $n/2$, then A is less efficient than B.

In the case of a parallel architecture, it is usual to assess theoretical time and processor bounds for a given algorithm.

For this purpose, it is convenient to introduce the idealised notion that during each time step unit (needed to perform a parallel algorithm) exactly one arithmetical operation can be carried out in parallel.

Let A be the algorithm under consideration. We then introduce the following notation:

$N(A)$ is the least number of processors required to obtain a maximum speed-up;

$T(A)$ is the number of time unit steps required when using $N(A)$ processors;

$T_p(A)$ is the number of time unit steps required when the number of processors available is restricted to $p < N(A)$.

Finally, we define an arithmetic expression as a string consisting of the four arithmetic operations $(+, -, *, /)$, left-hand brackets, right-hand brackets, "atoms" consisting of constants and variable operands.

The symbol $A\langle n\rangle$ will denote an arithmetic expression containing n atoms.

The time requested to evaluate $A\langle n\rangle$ using a single processor is equal to $n-1$ time units.

With an arbitrary number of processors, $A\langle n\rangle$ can be evaluated in $\log_2(n)$ time units.

Generally, the following result holds:

$$T(A\langle n\rangle) \geq \lceil \log_2(n) \rceil$$

where "$\lceil \log_2(n) \rceil$" denotes the smallest integer greater than or equal to $\log_2(n)$.

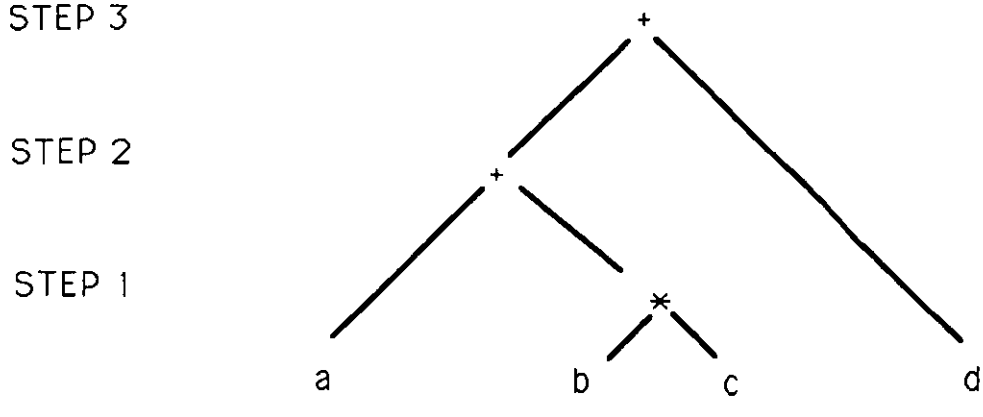
This result represents a lower limit for the parallel algorithm A (while using $N(A)$ processors).

On the other hand, we can notice that by exploiting properties such as associativity, commutativity and distributivity, it may be possible to transform the given expression $A\langle n\rangle$ into a form still equivalent but showing a better evaluation capability.

To best illustrate these concepts, let us consider the following example:

$$A\langle n\rangle = a + b * c + d \qquad \text{where } a,\ b,\ c,\ d \text{ are real numbers.}$$

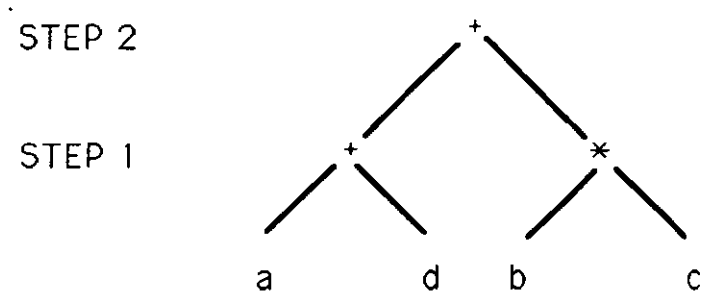
The analysis of this expression can be effected by the use of parse trees. The straightforward parse tree for $A\langle n\rangle$ involves three steps:



By exploitation of the commutative property of addition, $A\langle n\rangle$ can be rewritten as follows:

$$A\langle n\rangle = a + d + b * c.$$

Now a tree height of two is obtainable (see following figure):



To complete our considerations on complexity, we now mention a second result (proved by Kuck and Muraoka [19]), which gives an upper bound for a parallel algorithm:

if $A\langle n,d\rangle$ is an arithmetic expression, with n atoms and d nested brackets, then the properties of associativity and commutativity permit the transformation $A\langle n,d\rangle$ so that

$$T_p(A\langle n,d\rangle) \leq \lceil \log_2(n) \rceil + 2d + 1$$

with $p \leq \lceil n/2 - d \rceil$.

Other parameters

Speed-up and computational complexity give a way to measure the assessment of a parallel algorithm. For the same purpose, other parameters may be introduced.

If p processors are available and n is the dimension of the problem, the *efficiency* parameter is defined (by means of the speed-up factor) as

$$E_p(n) = \frac{S_p(n)}{p}.$$

$E_p(n)$ measures the utilisation of the parallel machine: the longer the processors are idle (or perform extra calculations due to the parallelization of the program), the smaller $E_p(n)$ becomes.

The *effectiveness* parameter $F_p(n)$ can be given as the ratio

$$F_p(n) = \frac{E_p(n) * S_p(n)}{T_s(n)}$$

and therefore it is a measure both of speed-up and efficiency. A parallel algorithm can then be regarded as effective if it maximises $F_p(n)$.

To conclude this section, we mention the fact that there are other aspects to be considered, such as stability and error analysis (rounding errors, propagation errors).

It may happen that parallel processing leads to numerically inferior results, but in most cases the parallel version of an algorithm actually leads to better results than the serial version [8].

3.3. Fundamentals of Parallel Numerical Analysis

It is of the greatest importance to recognise which problems already possess a parallel character, and which can be parallelized.

It is a common procedure to start with a serial algorithm and then convert it into a routine operating on vectors, the reason being that vector operations can be carried out in parallel.

This way of proceeding can be considered as a *first principle in the construction of parallel algorithms* (expecially for SIMD machines).

By applying this principle, the solution of an $n \times n$ triangular linear system, for example, can be achieved in about $3n$ steps, if n processors are available. The same system is solved on a serial computer with n^2 arithmetic operations. The speed-up in this hypotetical case is equal to

$$S_n(n) = \frac{n^2}{3n} = \frac{n}{3}$$

and the efficiency is $E_n(n) = 1/3$.

However usually n processors are not available.

If $k < n$ processors are available, $[n/k]$ parallel steps are necessary, corresponding to n serial steps ($[n/k]$ denotes the integer part of the real number n/k). Therefore, by using k processors the solution can be achieved in $O(n^2/k)$ steps. The speed-up is now,

$$S_k(n) = \frac{O(n^2)}{O(n^2/k)} = O(k)$$

and the efficiency is $E_k(n) = O(1)$.

A *second principle for the construction of parallel algorithms* is the method of "Vector Iteration".

This essentially consists of substituting an iterative parallel algorithm for a direct serial algorithm.

A simple application of this second principle is given by the triangular decomposition of a tridiagonal matrix suggested by Heller [15].

Let us denote:

$$A = \begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & & \ddots & \\ & & & & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & & & & 0 \\ l_2 & 1 & & & \\ & l_3 & 1 & & \\ & & \ddots & \ddots & \\ 0 & & & l_n & 1 \end{pmatrix} \qquad U = \begin{pmatrix} u_1 & c_1 & & & 0 \\ & u_2 & c_2 & & \\ & & u_3 & c_3 & \\ & & & \ddots & \ddots \\ & & & u_{n-1} & c_{n-1} \\ 0 & & & & u_n \end{pmatrix}$$

where

$$\begin{cases} u_1 = b_1 \\ u_i = b_i - a_i * c_{i-1}/u_{i-1} & i = 2, ..., n \end{cases}$$

and

$$l_i = a_i/u_{i-1} \qquad i = 2, ..., n.$$

The l_i quantities can be calculated in parallel.

The u_i calculations constitute a direct serial procedure (it is a linear recurrence formula of first order) that can be parallelized by the iteration:

$$\begin{cases} u_i^{(0)} = b_i \\ u_i^{(j)} = b_i - a_i * c_{i-1}/u_{i-1}^{(j-1)} & i = 1, 2, ..., n \end{cases}$$

where the symbol $u^{(j)}$ denotes the j^{th} iterate.

Obviously, this parallelization is only reasonable if the computer is able to perform one vector operation with vectors of n components faster than n scalar operations.

Furthermore, the number of iterations required must be significantly less than n.

A *third principle for the construction of parallel algorithms* is given by the method of "Recursive Doubling" (due to Kogge [17]).

In order to generally describe this method, consider a set of $N = 2^n$ elements:

$$M = \{m_1, m_2, m_3, ..., m_N\}$$

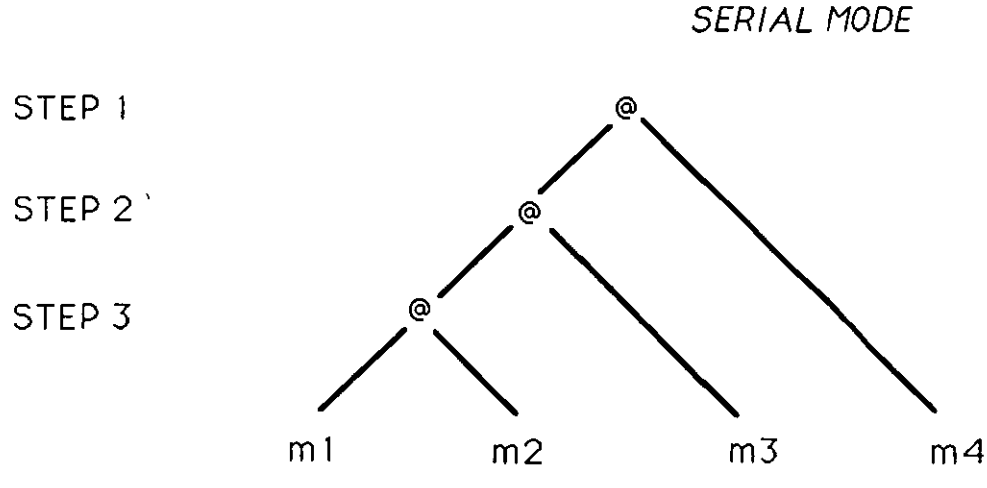
and consider an arbitrary associative operation @ defined on M.

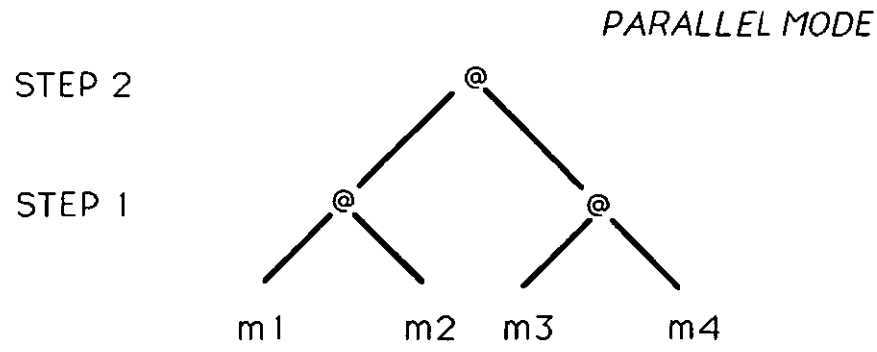
The expression:

$$A < N >= m_1 @ m_2 @ m_3 @ .... @ m_N$$

can be performed both serially and in parallel.

This can be illustrated by means of the following tree representations (suppose $N = 4$):





In general, recursive doubling requires $n = \log_2(N)$ parallel steps, while serial implementation requires $N - 1$ steps.

Both the vector iteration and recursive doubling methods (second and third principles) constitute a means of solving linear recurrence formulae of first and second order in parallel (such as those involved in the solution of tridiagonal systems).

4. The Wang Partitioning Method

4.1. Introduction

On the basis of efficient serial algorithms for the solution of tridiagonal linear systems there lie some fundamental numerical methods, such as:

1) the LU factorisation of a matrix with Gauss transformations (where L is lower triangular and U is upper triangular);

2) the method of Cramer;

3) the QR factorisation of a matrix (where Q is orthogonal and R is upper triangular).

In particular, by introducing parallelism into the first class of methods, parallel solution techniques are obtained, such as the odd-even cyclic reduction, the recursive doubling method, the Wang algorithm [1, 31].

We are mostly interested in the latter as a partition method to be compared with the Recursive Decoupling algorithm (see next chapter).

4.2. The Wang algorithm

Let us consider, then, an nxn system of tridiagonal linear equations:

$$b_i x_{i-1} + a_i x_i + c_i x_{i+1} = d_i \qquad i = 1, ..., n$$

with $x_0 = b_1 = c_n = x_{n+1} = 0$.

In matrix notation, the same system is represented as $A\mathbf{x} = \mathbf{d}$, that is

$$\begin{pmatrix} a_1 & c_1 & & & & 0 \\ b_2 & a_2 & c_2 & & & \\ & b_3 & a_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-1} & a_{n-1} & c_{n-1} \\ 0 & & & & b_n & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

A unique solution $\mathbf{x}$ exists for a given right-hand side $\mathbf{d}$ and nonsingular coefficient matrix A.

Given $n = p \cdot k$, where p is the number of processors available and k is an integer, the system under consideration is partitioned in $p \times p$ square blocks of dimension k. Each diagonal block is a $k \times k$ tridiagonal matrix, while all the subdiagonal (superdiagonal) blocks are $k \times k$ null matrices, except for one single non-zero element on its upper right (lower left) corner.

By applying elementary row transformations (Gaussian elimination) on all the p diagonal blocks simultaneously, the coefficient matrix A can be diagonalized in four steps.

Since the same partitioning process and Gaussian transformations have to be performed on the right-hand side vector $\mathbf{d}$, we will work on the augmented matrix $[A|\mathbf{d}]$.

Note that the hypothesis "k is an integer" is not essential in order to implement the Wang algorithm on an MIMD machine [18]: it only simplifies our notation, while illustrating the algorithm. Also note that we work under the assumption that the number of equations n is much larger than the number of processors p, and that pivoting (during the elementary row transformations) for numerical stability is not required.

For a clearer representation of the elimination pattern, we choose $n = 12$ and $p = 3$.

The initial augmented matrix is therefore partitioned as shown in the following figure 4.1.

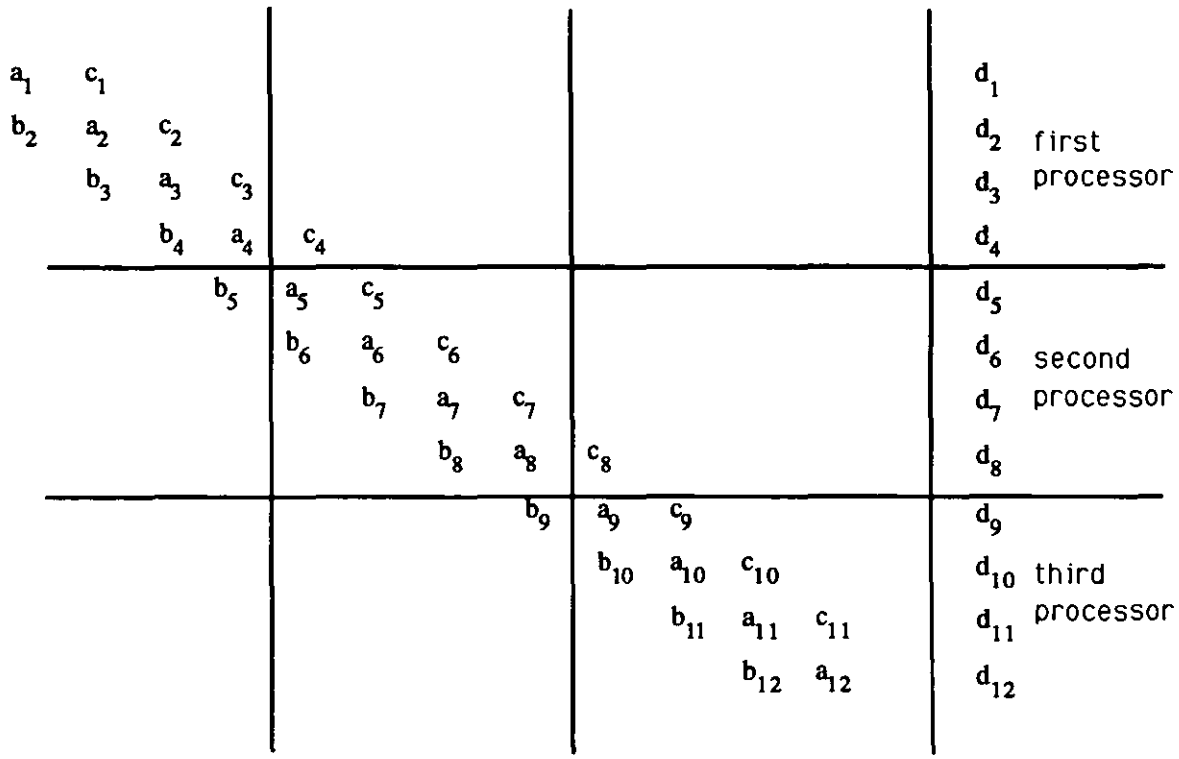


FIGURE 4.1. Initial augmented matrix.

After this partitioning phase, the first step (**step 1**) consists in applying elementary row transformations on all the p diagonal blocks simultaneously, so that each block is transformed into an upper triangular matrix.

This process creates fill-ins in the subdiagonal blocks: the rightmost column of each subdiagonal block is now completely filled (see elements f_i in figure 4.2).

During the second step (**step 2**), elimination continues on the superdiagonals of the diagonal blocks; the non-zero elements of the superdiagonal blocks are also eliminated. This process again creates fill-ins (the g_i elements in figure 4.2). Now the entire matrix is diagonal, except for the fill-ins (the so-called "fish-bone" form).

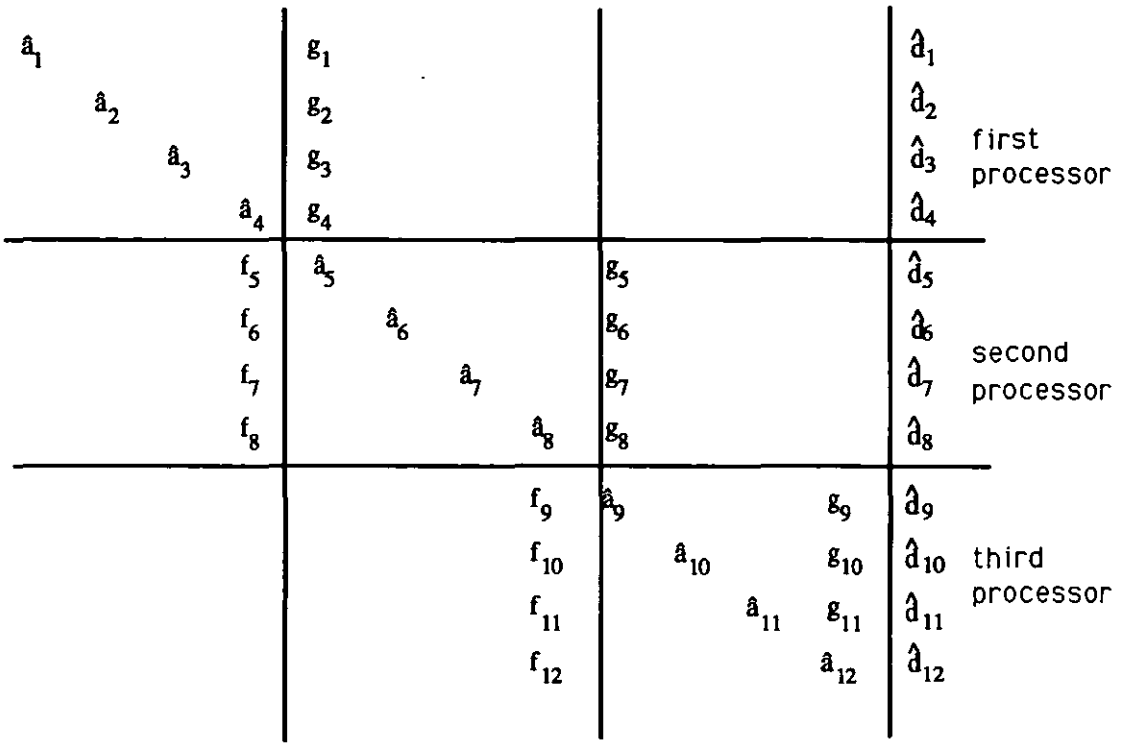


FIGURE 4.2. Augmented matrix after steps 1 and 2 (first way of performing Gaussian elimination).

The third and the fourth steps (**step 3** and **step 4**) consist of eliminating the non-zero elements below and above the main diagonal (Gauss-Jordan elimination) respectively; this process creates no new fill-ins. The coefficient matrix thus obtained is now diagonal and the solution can be computed by:

$$x_i = \frac{\hat{d}'_i}{\hat{a}'_i} \qquad i = 1, ..., n, \qquad \text{with} \quad \hat{a}'_i = 1 \ \forall i.$$

Alternatively, steps 3 and 4 can be substituted by the following procedure:

step 3') for $i = k,\ k+1,\ 2k,\ 2k+1,\ 3k,\ 3k+1, ...,\ p \cdot k,$ solve $2p-1$ tridiagonal equations of the form

$$f_i x_{i-1} + \hat{a}_i x_i + g_i x_{i+1} = \hat{d}_i$$

where $f_k = g_{p \cdot k} = 0$; at the end of this process the system is decomposed into p separate subsystems that can be solved independently;

step 4') calculate the remaining variables x_i by solving the independent subsystems.

Using a slightly more sophisticated elimination (Wang elimination), the augmented matrix obtained after the first two steps is of the form shown in the following figure:

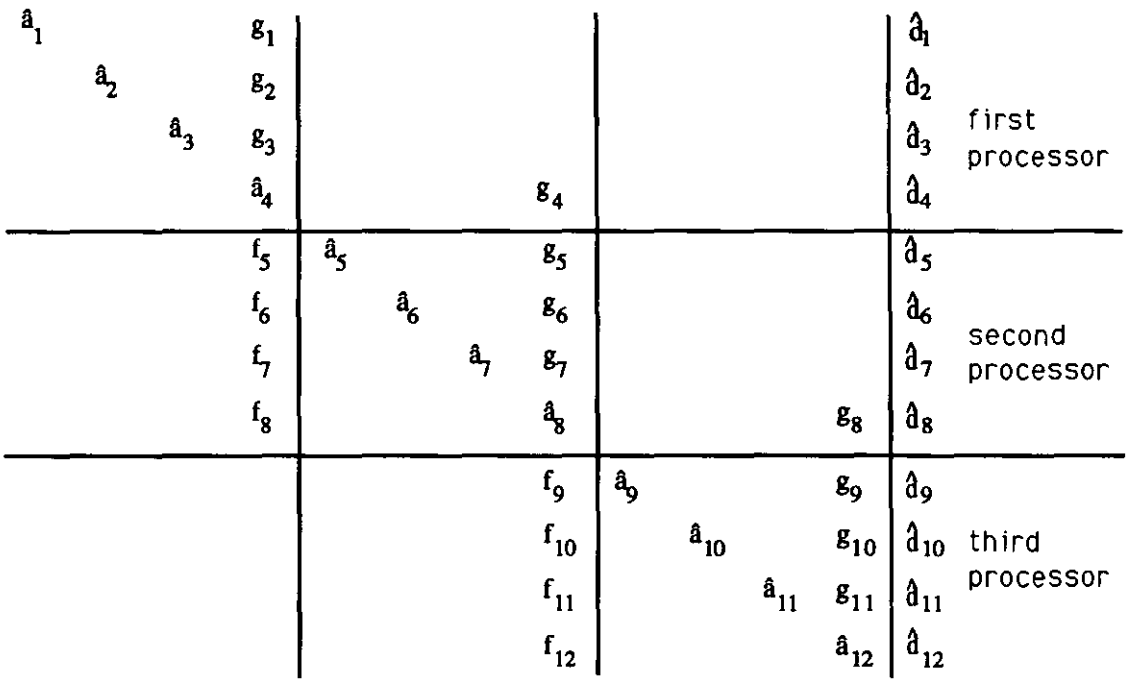


FIGURE 4.3. Augmented matrix after steps 1 and 2 (second way of performing Gaussian elimination).

In the above case of figure 4.3, only p tridiagonal equations need to be solved during the third phase; more precisely, the third step (*step 3'*) becomes the following:

for $i = k,\ 2k,\ 3k, ...,\ p \cdot k$, solve p tridiagonal equations of the form

$$f_i x_{i-1} + \hat{a}_i x_i + g_i x_{i+1} = \hat{d}_i$$

where $f_k = g_{p \cdot k} = 0$; at the end of this process the system is decomposed into p separate subsystems that can be solved independently.

The fourth step (*step 4'*)remains invariant.

The elimination process leading to the matrix form of figure 4.3 was first used by Wang as part of a different method for SIMD computer.

We now illustrate a version of the Wang algorithm suitable for an MIMD machine.

4.3. The Wang Fortran routine

Let **a**, **b**, **c** be the n-dimensional vectors containing the initial coefficient matrix A.

Let **x** and **d** be the unknown and known vectors, respectively, both of dimension n.

To perform the program's parallel sections, we need an additional array, **m**, of dimension $n + 1$.

Vectors **f** and **g**, both of dimension n, store the fill-in elements created by the algorithm.

Finally, five more work-arrays are utilised, each one of dimension equal to the number of processors available: let these arrays be called **aa**, **xx**, **dd**, **ff**, **gg**. They are used during a sequential part of the program, in order to correctly perform a call to the subroutine that solves the tridiagonal equations of *step 3'*.

The Wang routine starts with the initialisation of all the above mentioned variables and with the inputs concerning the number p of processors available and the number k of subsystems into which the entire system is partitioned.

As mentioned before, we set $n = p \cdot k$, where n, p and k are all integers.

In the case that n does not satisfy this condition, it is possible to add equations of the following type:

$$x_i = 1 \qquad i = n+1,\ n+2, ..., \ p \cdot k,$$

so that the order of the tridiagonal system is increased to dimension $p \cdot k$.

At this point, we are ready to perform the four steps of the Wang process.

STEP 1

p processors run in parallel.

We use the Doacross Parallel Programming directive to prepare a section of code for parallel execution, as shown below:

C\$Doacross share (k, p, f, b, m, a, c, d), local (j)

Do $i = 1,\ p$

if $(i \neq 1)$ $f_{(i-1)\cdot k+1} := b_{(i-1)\cdot k+1}$

Do $j = (i-1) \cdot k + 2,\ i \cdot k$

$m_j := b_j / a_{j-1}$

$a_j := a_j - m_j \cdot c_{j-1}$

$d_j := d_j - m_j \cdot d_{j-1}$

if $(i \neq 1)$ $f_j := -m_j \cdot f_{j-1}$

continue

continue

STEP 2

p processors run in parallel.

Again, we use the Doacross directive to perform the following Fortran code in parallel.

Note that there is no need for synchronization between steps 1 and 2, since the Doacross directive supports both the fork construct and the join construct (at the beginning and at the end of the loop, respectively).

C\$Doacross share (k, p, g, c, m, a, d, f), local (j)

Do $i = 1,\ p$

$g_{i \cdot k-1} := c_{i \cdot k-1}$

Do $j = i \cdot k - 2,\ (i-1) \cdot k + 1,\ -1$

$m_j := c_j / a_{j+1}$

$g_j := -m_j \cdot g_{j+1}$

$d_j := d_j - m_j \cdot d_{j+1}$

if $(i \neq 1)$ $f_j := f_j - m_j \cdot f_{j+1}$

continue

if $(i \neq 1)$ then

$m_i := c_{(i-1) \cdot k} / a_{(i-1) \cdot k+1}$

$g_{(i-1) \cdot k} := -m_i \cdot g_{(i-1) \cdot k+1}$

$a_{(i-1) \cdot k} := a_{(i-1) \cdot k} - m_i \cdot f_{(i-1) \cdot k+1}$

$d_{(i-1) \cdot k} := d_{(i-1) \cdot k} - m_i \cdot d_{(i-1) \cdot k+1}$

endif

continue

STEP 3

The third phase of the Wang method is performed serially and consists of solving p tridiagonal equations:

$$f_i x_{i-k} + a_i x_i + g_i x_{i+k} = d_i \qquad for\ i = k,\ p \cdot k,\ k \qquad (f_k = g_{p \cdot k} = 0).$$

This is done by calling a subroutine which solves the i^{th} equation, giving the solution x_i.

In order to avoid modifications in the vectors **a**, **x**, **d**, **f** and **g**, the subroutine call is preceded and followed by saving the components concerned in the work-arrays **aa**, **xx**, **dd**, **ff** and **gg** (respectively).

STEP 4

p processors run in parallel.

The Doacross directive is utilized to prepare the loops for parallel execution.

C\$Doacross share (k, p, d, f, g, a, x), local (j)

Do $i = 1,\ p$

Do $j = (i-1) \cdot k + 1,\ i \cdot k - 1$

if $(i \neq 1)$ then

$$x_j := (d_j - f_j \cdot x_{(i-1) \cdot k} - g_j \cdot x_{i \cdot k}) / a_j$$

else

$$x_j := (d_j - g_j \cdot x_{i \cdot k}) / a_j$$

continue

continue

At this point, the complete solution vector **x** is obtained. Again, there is no need for synchronization at the end of step 4, since the Doacross statement provides it.

4.4. Numerical Experiments and Remarks

In this paragraph numerical results are reported, concerning the solution of tridiagonal linear systems by means of the Wang routine, on the Balance 8000 parallel machine.

The following tables group together the execution timing (both for the sequential and the parallel versions of the Wang algorithm), the experimental speed-up (to be compared with the expected speed-up) and the efficiency parameters.

The maximum error E_{max}, average error E_{av} and maximum relative error E_r obtained are also presented, in order to study the degree of accuracy reached by the method. These error measurement have been calculated according to the following formulae:

$$\begin{aligned} E_{max} &= \|x - \tilde{x}\|_\infty \\ E_{av} &= \frac{\sum_{i=1}^{n} |x_i - \tilde{x}_i|}{n} \\ E_r &= \frac{\|x - \tilde{x}\|_\infty}{\|x\|_\infty} \end{aligned} \tag{4.4}$$

where $\mathbf{x} = (x_1, ..., x_i, ..., x_n)^T$ is the exact solution of the system

and $\tilde{\mathbf{x}} = (\tilde{x}_1, ..., \tilde{x}_i, ..., \tilde{x}_n)^T$ is the calculated solution.

All the results shown are related to two test tridiagonal systems (with known solution), whose coefficient matrices satisfy the condition:

$$b_i \geq a_i + c_i \quad \forall i = 1, 2, .., n$$

where $b_1, ..., b_i, ..., b_n$ are the elements on the main diagonal;

$a_1, ..., a_i, ..., a_n$ are the elements on the sub-diagonal (with $a_1 = 0$);

$c_1, ..., c_i, ..., c_n$ are the elements on the super-diagonal (with $c_n = 0$).

The two example systems chosen are as described below.

First test system

The first tridiagonal linear system considered [14] is

$$\begin{pmatrix} 2 & -1 & & & & 0 \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ 0 & & & & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{4.5}$$

whose exact solution is an n-dimensional vector $\mathbf{x}$ with components:

$$x_i = \frac{n+1-i}{n+1} \quad \forall i = 1, .., n.$$

Second test system

The coefficient matrix in this second example is a tridiagonal matrix of the form:

$$\begin{pmatrix} T_8 & & & \\ & T_8 & & \\ & & \ddots & \\ & & & T_8 \end{pmatrix}$$

where each submatrix T_8 is an 8×8 tridiagonal matrix

$$T_8 = \begin{pmatrix} 2 & -1 & & & & & & 0 \\ -3 & 5 & -2 & & & & & \\ & -2 & 3 & -1 & & & & \\ & & -2 & 4 & -1 & & & \\ & & & -1 & 4 & -3 & & \\ & & & & -4 & 6 & -1 & \\ & & & & & -7 & 8 & -1 \\ 0 & & & & & & -1 & 3 \end{pmatrix} \tag{4.6}.$$

According to the block structure of the coefficient matrix, the known vector $\mathbf{d}$ has the form:

$$\mathbf{d} = (\mathbf{d_8}, \mathbf{d_8}, ..., \mathbf{d_8})^T$$

where each subvector $\mathbf{d_8}$ has dimension 8 and components:

$$\mathbf{d_8} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 2 \end{pmatrix} \qquad (4.7).$$

The exact solution vector to this second test system is an n-dimensional vector $\mathbf{x}$, whose components are all equal to 1.

If we choose the dimension $n = 16$, for example, the tridiagonal coefficient matrix will be (in this second example):

$$\begin{pmatrix}
2 & -1 & & & & & & & & & & & & & & \\
-3 & 5 & -2 & & & & & & & & & & & & & \\
 & -2 & 3 & -1 & & & & & & & & & 0 & & & \\
 & & -2 & 4 & -1 & & & & & & & & & & & \\
 & & & -1 & 4 & -3 & & & & & & & & & & \\
 & & & & -4 & 6 & -1 & & & & & & & & & \\
 & & & & & -7 & 8 & -1 & & & & & & & & \\
 & & & & & & -1 & 3 & 0 & & & & & & & \\
 & & & & & & & 0 & 2 & -1 & & & & & & \\
 & & & & & & & & -3 & 5 & -2 & & & & & \\
 & & & & & & & & & -2 & 3 & -1 & & & & \\
 & & & & & & & & & & -2 & 4 & -1 & & & \\
 & & & & & & & & & & & -1 & 4 & -3 & & \\
 & & 0 & & & & & & & & & & -4 & 6 & -1 & \\
 & & & & & & & & & & & & & -7 & 8 & -1 \\
 & & & & & & & & & & & & & & -1 & 3
\end{pmatrix}$$

and the corresponding known vector $\mathbf{d}$ will be:

$$(1,0,0,1,0,1,0,2,1,0,0,1,0,1,0,2)^T.$$

The solution vector $\mathbf{x}$ we are looking for, in this case, is:

$$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)^T.$$

Tables 4.8 and 4.9 show the execution timings obtained when running the Wang routine on the two test examples mentioned. The speed-up and efficiency parameters are also calculated and reported in the same tables.

In these experiments the Parallel System overheads introduced by the operating system proved to be too great to ensure a speed-up of the algorithm.

Consequently, it was necessary to evaluate *reclaimed times* for the Wang algorithm; that is to say, the elapsed execution times minus the cost of forking child processes and child page table build up (child processes do not automatically inherit the parent's page table when created) was used in evaluating the speed-up.

The notation adopted in all the table is as follows:

n: is the dimension of the tridiagonal linear system under consideration;

q: is the exponent (power of 2) so that $n = 2^q$;

p: is the number of processors used.

The accuracy results are given in tables 4.10 and 4.11 for the parallel version and the sequential version of the Wang algorithm respectively.

Some slight discrepancy in the accuracy results do occur, and this is probably due to favourable partitioning which reduces rounding errors.

Note that these results only concern the solution of the first test system, since in the case of the second test system 100 % accuracy is obtained. This is probably due to the fact that the solution vector $\mathbf{x}$ (in the second example) can be regarded as having all integer components (they are all equal to 1). Therefore, no rounding error is involved.

TABLE 4.8

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=128 q=7 p=4	.02650	.0280	.9464	1.8824	.2366
n=256 q=8 p=4	.05200	.0330	1.5758	1.9394	.3939
n=512 q=9 p=4	.10350	.0600	1.7250	1.9692	.4313
n=1024 q=10 p=4	.20600	.1000	2.0600	1.9845	.5150

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=128 q=7 p=8	.02650	.0210	1.2619	3.2000	.1577
n=256 q=8 p=8	.05200	.0280	1.8571	3.5556	.2321
n=512 q=9 p=8	.10350	.0420	2.4643	3.7647	.3080

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Wang routine to solve the first example system (4.5).

TABLE 4.9

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=128 q=7 p=4	.025000000	.032999990	.7576	1.8824	.1894
n=256 q=8 p=4	.050000000	.042000010	1.1905	1.9394	.2976
n=512 q=9 p=4	.099500030	.059000000	1.6864	1.9692	.4216
n=1024 q=10 p=4	.199000000	.100000000	1.9900	1.9845	.4975

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=128 q=7 p=8	.025000000	.02150	1.1628	3.2000	.1453
n=256 q=8 p=8	.050000000	.02800	1.7857	3.5556	.2232
n=512 q=9 p=8	.099500030	.05200	1.9135	3.7647	.2392

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Wang routine to solve the second example system (4.6) & (4.7).

TABLE 4.10

	Maximum Error E_{max}	Average Error E_{av}	Maximum Relative Error E_r
n=128 q=7 p=4	.0000018477	.0000011250	.0000018621
n=256 q=8 p=4	.0000088215	.0000054081	.0000088560
n=512 q=9 p=4	.0000028610	.0000007448	.0000028666
n=1024 q=10 p=4	.0001227856	.0000744966	.0001229055

	Maximum Error E_{max}	Average Error E_{av}	Maximum Relative Error E_r
n=128 q=7 p=8	.0000030994	.0000019837	.0000031236
n=256 q=8 p=8	.0000064969	.0000042093	.0000065223
n=512 q=9 p=8	.0000324249	.00000209412	.0000324882

Accuracy results obtained when using the parallel version of the Wang routine to solve the first example system (4.5).

TABLE 4.11

	Maximum Error E_{max}	Average Error E_{av}	Maximum Relative Error E_r
n=128 q=7	.0000013709	.0000006365	.0000013816
n=256 q=8	.0000044703	.0000019153	.0000044878
n=512 q=9	.0000141859	.0000066210	.0000142136
n=1024 q=10	.0001955032	.0001060939	.0001956941

Accuracy results obtained when using the sequential version of the Wang routine to solve the first example system (4.5).

Note.
In tables 4.8,4.9, obtained and expected speed-ups are given according to the following formulae:

$$Obtained S_p(n) = T_s/T_p$$

$$Expected S_p(n) = n/(2k+p)$$

where n is the problem dimension, p is the number of processors, $k = n/p$.
The Efficiency parameter is $E_p(n) = S_p(n)/p$.

5. The Recursive Decoupling Method

5.1. Introduction to the Recursive Decoupling Method

In this chapter we describe a new tridiagonal equation solver, based on a rank-one updating strategy and the repeated partitioning of the system matrix into 2x2 submatrices. On these bases, a recursive decoupling method is developed, which operates on the tridiagonal linear system, enabling the solution to be expressed in explicit form and solved independently on a multiprocessor system. We will show, in fact, that the Recursive Decoupling Method is intrinsically parallel and can be implemented as an efficient parallel algorithm [5, 6].

5.2. The Partitioning Process

We consider a set of n linear equations in n unknowns

$$A\mathbf{u} = \mathbf{d} \tag{5.1}$$

where A is an nxn tridiagonal matrix of the form

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & & & & \\ & 0 & & & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \quad \text{with} \quad b_i \geq a_i + c_i \ , \quad \forall i = 1, 2, .., n \tag{5.2}$$

We denote $\mathbf{d}$ and $\mathbf{u}$ as the n-dimensional known and unknown vectors respectively.

To illustrate the algorithm, we assume that n is an integer power of 2, i.e. $n = 2^q$ (with $m = n/2 = 2^{q-1}$). This assumption is not restrictive: the method can be generalized. The choice $n = 2m$ simplifies our notation.

The initial coefficient matrix A is now rearranged into the following partitioned form

$$\begin{pmatrix} e_1 & c_1 & & & & & \\ a_2 & e_2 & & & 0 & & \\ & & e_3 & c_3 & & & \\ & & a_4 & e_4 & & & \\ & & & & \ddots & & \\ & 0 & & & & e_{n-1} & c_{n-1} \\ & & & & & a_n & e_n \end{pmatrix} + \sum_{j=1}^{m-1} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}^{(j)} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix}^{(j)T} \tag{5.3}$$

where

$$\begin{aligned} e_1 &= b_1 \\ e_{2j-1} &= b_{2j-1} - c_{2j-2} \qquad \text{when} \quad j = 2, \ldots, m \\ e_{2j} &= b_{2j} - a_{2j+1} \qquad \text{when} \quad j = 1, \ldots, m-1 \\ e_n &= b_n \end{aligned} \tag{5.4}$$

and where vectors $\mathbf{x}^{(j)}$, $\mathbf{y}^{(j)}$ have only non-zero elements in the $2j^{th}$ and $(2j+1)^{th}$ positions,

i.e.

$$\begin{cases} x_k = 1 & \text{when} \quad k = 2j, 2j+1 \\ x_k = 0 & \text{otherwise} \end{cases}$$

$$\begin{cases} y_k = a_{2j+1} & \text{when} \quad k = 2j \\ y_k = c_{2j} & \text{when} \quad k = 2j+1 \\ y_k = 0 & \text{otherwise} \end{cases}$$

that is

$$\mathbf{x}^{(j)} = (0, \ldots, 0, 1, 1, 0, \ldots, 0)^T \tag{5.5}$$

$$\mathbf{y}^{(j)} = (0, \ldots, 0, a_{2j+1}, c_{2j}, 0, \ldots, 0)^T \tag{5.6}$$

with j ranging from 1 to $m-1$ and $n = 2m$.

In matrix notation, therefore, the above partitioning of the matrix A can be represented as

$$A = J + \sum_{j=1}^{m-1} \mathbf{x}^{(j)}\mathbf{y}^{(j)T} \tag{5.7}$$

where J is a block diagonal matrix of the form

$$J = \begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_m \end{pmatrix} \tag{5.8}$$

Each block in J is a 2x2 submatrix J_i of the following type

$$J_i = \begin{pmatrix} e_{2i-1} & c_{2i-1} \\ a_{2i} & e_{2i} \end{pmatrix} \quad \text{with} \quad i = 1, 2, \ldots, m \tag{5.9}$$

The elements of J_i are defined as in (5.4).

The basic idea, underlying the choice of this particular partitioning, is given by the Sherman-Morrison formula.

Suppose that we have computed the inverse matrix J^{-1} for some matrix J of dimension nxn. Then, suppose that J is modified into a matrix A as follows

$$A = J + \mathbf{x}\mathbf{y}^T \tag{5.10}$$

where $\mathbf{x}$, $\mathbf{y}$ are n-dimensional vectors.

According to Sherman-Morrison [27], the inverse A^{-1} can be computed as

$$A^{-1} = J^{-1} - \alpha(J^{-1}\mathbf{x})(\mathbf{y}^T J^{-1}) \qquad \text{where} \quad \alpha = 1/(1 + \mathbf{y}^T J^{-1}\mathbf{x}) \tag{5.11}$$

To compute the new inverse directly would cost $O(n^3)$ arithmetic operations, while the use of formula (5.11) only implies $O(n^2)$ operations.

The Sherman-Morrison formula also applies if linear equations are being solved, so that the solution of $J\mathbf{u}' = \mathbf{d}$ must be converted to the solution of $A\mathbf{u} = \mathbf{d}$.

From formula (5.10), we have

$$\begin{aligned} \mathbf{u} = A^{-1}\mathbf{d} &= (J^{-1} - \alpha(J^{-1}\mathbf{x})(\mathbf{y}^T J^{-1}))\mathbf{d} \\ &= J^{-1}\mathbf{d} - \alpha(J^{-1}\mathbf{x})(\mathbf{y}^T J^{-1})\mathbf{d} \end{aligned} \tag{5.12}$$

Suppose we have a routine that can solve linear systems involving matrices J or J^{-1}. The solution of the modified system $A\mathbf{u} = \mathbf{d}$ can then be obtained from the following sequence

(i) solve $J\mathbf{u}' = \mathbf{d}$ for $\mathbf{u}'$, so that $\mathbf{u}' = J^{-1}\mathbf{d}$ is known;

(ii) solve $J\mathbf{w} = \mathbf{x}$ for $\mathbf{w}$, so that $\mathbf{w} = J^{-1}\mathbf{x}$ is known;

(iii) solve $J^T\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$, so that $\mathbf{z}^T = \mathbf{y}^T J^{-1}$ is known;

(iv) form $\alpha = 1/(1 + \mathbf{y}^T\mathbf{w})$;

(v) form $\mathbf{u} = \mathbf{u}' - \alpha\mathbf{w}\mathbf{z}^T\mathbf{d}$, the solution of $A\mathbf{u} = \mathbf{d}$.

This process requires back-substitutions and inner-products, so that the cost is only $O(n)$ operations. It also avoids the explicit computation of the inverse matrix.

In order to apply the Sherman-Morrison formula, therefore, all we need to know is a matrix J which only differs by a few elements from our coefficient matrix A and whose inverse J^{-1} is known.

If we now go back to our original problem of solving the tridiagonal system $A\mathbf{u} = \mathbf{d}$ and consider again the partitioning formula (5.7), we can notice the similarities with the Sherman-Morrison expression. Our partitioning form (5.7) can be considered as a recursive application of formula (5.10). Besides, our coefficient matrix A only differs by a few elements from a block diagonal matrix J, whose blocks J_i are of dimension 2x2 and are therefore immediately invertible.

For every index i ranging from 1 to m, in fact, we have

$$J_i^{-1} = \frac{1}{\Delta_i} \begin{pmatrix} e_{2i} & -c_{2i-1} \\ -a_{2i} & e_{2i-1} \end{pmatrix} \qquad \text{with} \quad \Delta_i = e_{2i-1}e_{2i} - a_{2i}c_{2i-1} \tag{5.13}$$

It is then possible for us to use the Sherman-Morrison formula.

In the next paragraph, we show how a process, based on similar ideas to the ones described in (5.12), have been developed into the Recursive Decoupling Method.

5.3. The Recursive Decoupling Process

Given the tridiagonal linear system $A\mathbf{u} = \mathbf{d}$, where A is rewritten as in (5.7), we now want to partition the two vectors $\mathbf{u}$ and $\mathbf{d}$ in an analogous way to the matrix J.

To this purpose, we consider vectors $\mathbf{u}$ and $\mathbf{d}$ to be written as follows

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \\ u_3 \\ u_4 \\ \\ \vdots \\ \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} \mathbf{u}_1 \\ \\ \mathbf{u}_2 \\ \\ \vdots \\ \\ \mathbf{u}_m \end{pmatrix} \qquad \mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \\ d_3 \\ d_4 \\ \\ \vdots \\ \\ d_{n-1} \\ d_n \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \\ \mathbf{d}_2 \\ \\ \vdots \\ \\ \mathbf{d}_m \end{pmatrix} \tag{5.14}$$

We can now derive the solution of system (5.1) by applying the rank-one updating procedure of Sherman-Morrison recursively to (5.7). The result is

$$\begin{aligned} &\mathbf{u} = (J + \sum_{j=1}^{m-1} \mathbf{x}^{(j)}\mathbf{y}^{(j)T})^{-1}\mathbf{d} = (J^{-1} - \alpha_j J^{-1} \sum_{j=1}^{m-1} \mathbf{x}^{(j)}\mathbf{y}^{(j)T}J^{-1})\mathbf{d} \\ &\text{with } \alpha_j = 1/(1 + \sum_{j=1}^{m-1} \mathbf{y}^{(j)T}J^{-1} \sum_{j=1}^{m-1} \mathbf{x}^{(j)}) \\ &\text{and } j = 1, 2, \ldots, m-1 \end{aligned} \tag{5.15}$$

By denoting $J^{-1}\mathbf{d} = \mathbf{u}'$ and $J^{-1}\mathbf{x}^{(j)} = \mathbf{g}^{(j)}$, the expressions (5.15) can be simplified to

$$\begin{aligned} &\mathbf{u} = (I - \alpha \mathbf{g}^{(j)} \mathbf{y}^{(j)T})^{-1} \mathbf{u}' \\ &\text{with } \alpha = 1/(1 + \mathbf{y}^{(j)T} \mathbf{g}^{(j)}) \\ &\text{and } j = 1, 2, \ldots, m-1 \end{aligned} \tag{5.16}$$

The expressions (5.16) give a description of the rank-one updating procedure.

Once we have applied the partitioning process (5.7), in order to obtain the final solution $\mathbf{u}$, we then need

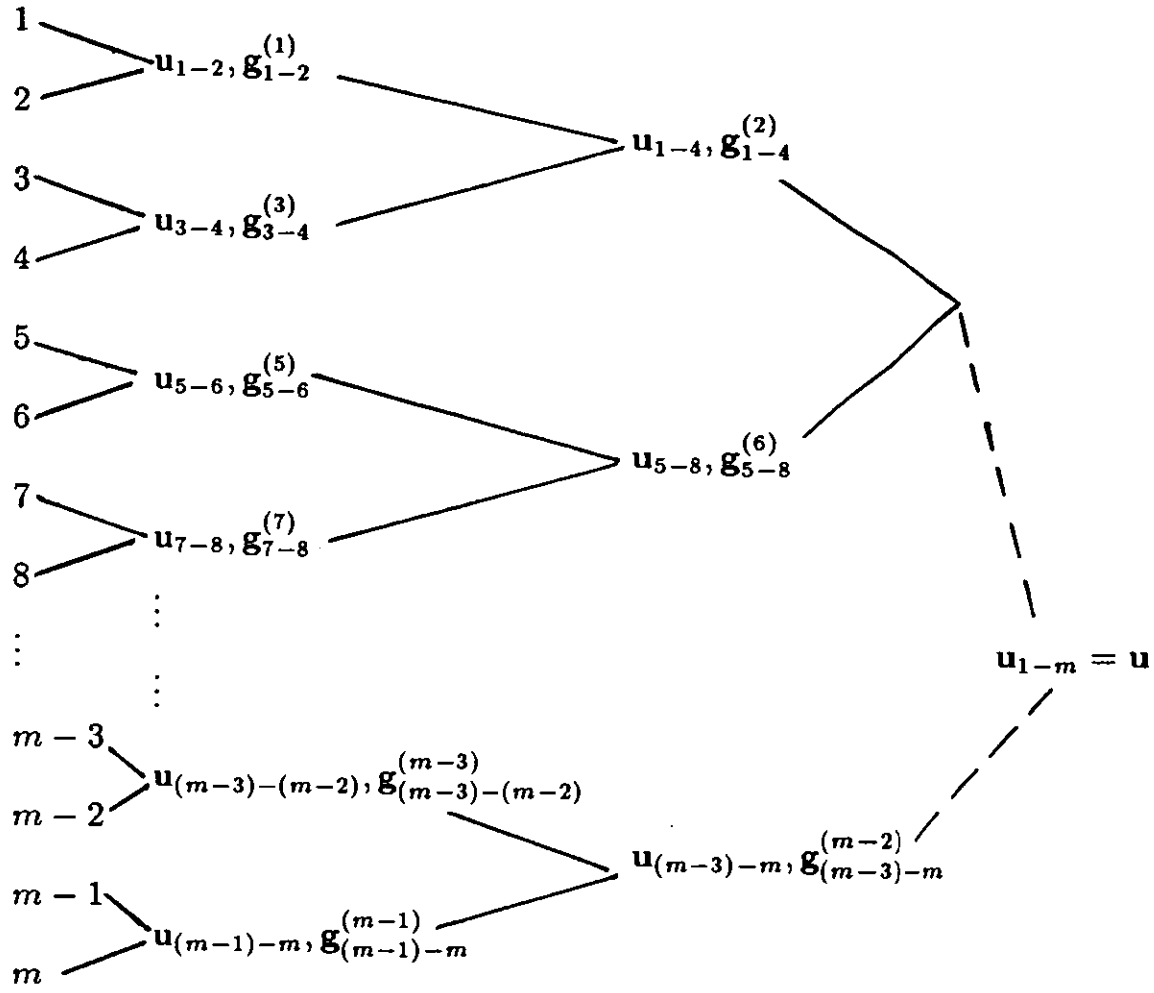
- to start with an approximated solution vector $\mathbf{u}'$ given by $\mathbf{u}' = J^{-1}\mathbf{d}$;

- to calculate vectors $\mathbf{g}^{(j)}$ from $\mathbf{g}^{(j)} = J^{-1}\mathbf{x}^{(j)}$;

- to perform the rank-one updating procedure (5.16).

In particular, the updating step implies the recursive use of vectors $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$ and the recursive updating of vector $\mathbf{u}$ and vectors $\mathbf{g}^{(j)}$.

Note that, for each index j ranging from 1 to $m-1$, the form of vectors $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$ are such that they only contain 2 non-zero elements. Therefore the expressions (5.16) can be calculated independently and recursively using a *Parallel Fan-in Algorithm*, as follows



(5.17)

The notation adopted in figure (5.17) uses the subscript index of vectors $\mathbf{u}$ and $\mathbf{g}^{(j)}$ to indicate the number of components involved in the current calculation. For example, the writing $\mathbf{g}^{(1)}_{1-2}$ means that we are considering components 1 and 2 of vector $\mathbf{g}^{(1)}$, while $\mathbf{g}^{(6)}_{5-8}$ means that we are considering components 5, 6, 7, 8 of vector $\mathbf{g}^{(6)}$.

More specifically:

$\mathbf{u}_{i-(i+k)}$ is a vector of $2k$ components, namely components $2i, 2i+1, ..., 2i+2k$ of vector $\mathbf{u}$

$\mathbf{g}^{(j)}_{i-(i+k)}$ is a vector of $2k$ components, namely components from $2i$ to $2i+2k$ of vector $\mathbf{g}^{(j)}$.

Therefore:

- at the 1^{st} level of the above tree structure, vectors $\mathbf{u}_{i-(i+1)}$ and $\mathbf{g}^{(j)}_{i-(i+1)}$ are composed of 4 components, with the index i ranging from 1 to m and $j = 1, 3, .., m-1$;

- at the 2^{nd} level, vectors $\mathbf{u}_{i-(i+3)}$ and $\mathbf{g}^{(j)}_{i-(i+3)}$ are composed of 8 components, with the index i ranging from 1 to m and $j = 2, 6, .., m-2$.

- at the 3^{rd} level, vectors $\mathbf{u}_{i-(i+7)}$ and $\mathbf{g}^{(j)}_{i-(i+7)}$ are composed of 16 elements, with the index i ranging from 1 to m and $j = 4, 12, .., m-4$.

These observations apply in an analogous way to each level. The last level gives the final solution $\mathbf{u} = \mathbf{u}_{1-m}$, which is a vector of $2m = n$ components.

The tree structure depicted in (5.17) enables as many as possible of the matrix updating strategies to be performed in parallel. The calculations of vectors $\mathbf{u}_{i-(i+k)}$ and $\mathbf{g}^{(j)}_{i-(i+k)}$ are, in fact, non overlapping and independent of each other.

The Fan-in graph represented in figure (5.17) has a depth equal to $\log_2 m$, and it is composed of $\log_2 m$ levels. The whole algorithm, implementing the Recursive Decoupling Method, exhibits an average degree of parallelism of $O(m/\log_2 m)$. This result only applies to systems of order $n = 2^q$ and is likely to be degraded for systems whose order n is not an integer power of 2, due to imperfect load balancing.

5.4. The Recursive Decoupling Algorithm

In this section we present the solution procedure in algorithmic form, when the coefficient matrix A is partitioned as in (5.3) and the unknown and known vectors $\mathbf{u}$ and $\mathbf{d}$ are partitioned as in (5.14). As already said, we discuss the case for $n = 2^q$, but the process is equally valid for n not equal to a power of 2 and can easily be adapted.

We will refer to the above mentioned partitioning of matrix A and vectors $\mathbf{u}$ and $\mathbf{d}$ as the *Preliminary Stage* or *Pre-stage*. After this Pre-stage, the solution routine is formulated into three different sections, respectively called *Stage* 1, *Stage* 2 and *Stage* 3.

Stage 1.

This first stage of the algorithm consists of finding the solution of $J\mathbf{u} = \mathbf{d}$, that is obtaining

$$\mathbf{u} = \begin{pmatrix} J_1^{-1} & & & \\ & J_2^{-1} & & \\ & & \ddots & \\ & & & J_m^{-1} \end{pmatrix} \mathbf{d} \tag{5.18}$$

This is equivalent to solving m systems of the form

$$\begin{pmatrix} u_{2i-1} \\ u_{2i} \end{pmatrix} = J_i^{-1} \begin{pmatrix} \mathbf{d}_{2i-1} \\ \mathbf{d}_{2i} \end{pmatrix} \tag{5.19}$$

Since we know the expressions for matrices J_i^{-1} (see formulae (5.13)), the solution $\mathbf{u}$ of system (5.18) can be explicitly expressed as

$$\mathbf{u} = \begin{pmatrix} (e_2 d_1 - c_1 d_2)/\Delta_1 \\ (-a_2 d_1 + e_1 d_2)/\Delta_1 \\ \vdots \\ (e_{2i} d_{2i-1} - c_{2i-1} d_{2i})/\Delta_i \\ (-a_{2i} d_{2i-1} + e_{2i-1} d_{2i})/\Delta_i \\ \vdots \\ (e_{2m} d_{2m-1} - c_{2m-1} d_{2m})/\Delta_m \\ (-a_{2m} d_{2m-1} + e_{2m-1} d_{2m})/\Delta_m \end{pmatrix} \tag{5.20}$$

Note that each one of the m subsystems (5.19) can be solved independently on a multiprocessor.

Stage 2.

In an analogous way to the previous section, the second stage consists of finding the solution of $J\mathbf{g}^{(j)} = \mathbf{x}^{(j)}$ for $j = 1, 2, .., m-1$, that is obtaining

$$\mathbf{g}^{(j)} = \begin{pmatrix} J_1^{-1} & & & \\ & J_2^{-1} & & \\ & & \ddots & \\ & & & J_m^{-1} \end{pmatrix} \mathbf{x}^{(j)} \tag{5.21}$$

for $j = 1, 2, .., m-1$.

As before, this is equivalent to solving m systems of the form

$$\begin{pmatrix} g_{2i-1}^{(j)} \\ g_{2i}^{(j)} \end{pmatrix} = J_i^{-1} \begin{pmatrix} x_{2i-1}^{(j)} \\ x_{2i}^{(j)} \end{pmatrix} \tag{5.22}$$

where j ranges from 1 to $m-1$.

Again, since matrices J_j^{-1} are known, we can express the solution $\mathbf{g}^{(j)}$ of each one of the $m-1$ systems (5.21) as follows

$$\mathbf{g}^{(j)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ (e_{2i} - c_{2i-1})/\Delta_i \\ (-a_{2i} + e_{2i-1})/\Delta_i \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{5.23}$$

where the only two non-zero elements appear in the $(2j)^{th}$ and $(2j+1)^{th}$ positions.

Furthermore, the $m-1$ systems (5.21) can also be evaluated in parallel, by applying the structure shown in the following figure:

$$\begin{pmatrix} g_1^{(1)} \\ g_2^{(1)} \end{pmatrix} = J_1^{-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} g_3^{(1)} & g_3^{(2)} \\ g_4^{(1)} & g_4^{(2)} \end{pmatrix} = J_2^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} g_5^{(2)} & g_5^{(3)} \\ g_6^{(2)} & g_6^{(3)} \end{pmatrix} = J_3^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad (5.24)$$

$$\vdots$$

$$\begin{pmatrix} g_{n-3}^{(m-2)} & g_{n-3}^{(m-1)} \\ g_{n-2}^{(m-2)} & g_{n-2}^{(m-1)} \end{pmatrix} = J_{m-1}^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} g_{n-1}^{(m-1)} \\ g_n^{(m-1)} \end{pmatrix} = J_m^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The process described in figure (5.24) relies on the fact that, before starting the updating Stage 3, the vectors $\mathbf{g}^{(j)}$ contain only two non-zero elements, since they are derived from vectors $\mathbf{x}^{(j)}$.

Stage 3.

Finally, during this last stage, vectors $\mathbf{u}$ and $\mathbf{g}^{(j)}$ are updated, by recursively using the Sherman-Morrison formula.

The rank-one updating step procedure can be described as follows

$$
\begin{aligned}
&\text{for } k = 1, 2, .., q-1 \\
&\quad \text{for } j = 2^{k-1}, 2^{q-1} - 2^{k-1}, 2^k \\
&\quad\quad \alpha_j = 1/(1 + \mathbf{y}^{(j)T}\mathbf{g}^{(j)}) \\
&\quad\quad \mathbf{u} = (I - \alpha_j \mathbf{g}^{(j)}\mathbf{y}^{(j)T})^{-1}\mathbf{u} \\
&\quad\quad \text{for } i = 2^k, 2^{q-1} - 2^k, 2^{k+l} \\
&\quad\quad\quad \mathbf{g}^{(i)} = (I - \alpha_j \mathbf{g}^{(j)}\mathbf{y}^{(j)T})^{-1}\mathbf{g}^{(i)} \\
&\quad\quad \text{end} \\
&\quad \text{end} \\
&\text{end}
\end{aligned}
\tag{5.25}
$$

The final solution is obtained and stored in vector $\mathbf{u}$.

5.5. An Analytical Example

Before describing the Fortran routine which implements the Recursive Decoupling Algorithm, we give an example to illustrate the solution strategy.

Consider a 16x16 tridiagonal linear system, whose coefficient matrix A is given by

$$A = \begin{pmatrix}
b_1 & c_1 & & & & & & & & & & & & & & \\
a_2 & b_2 & c_2 & & & & & & & & & & & & & \\
 & a_3 & b_3 & c_3 & & & & & & & & & & & & \\
 & & a_4 & b_4 & c_4 & & & & & & & & & & & \\
 & & & a_5 & b_5 & c_5 & & & & & & 0 & & & & \\
 & & & & a_6 & b_6 & c_6 & & & & & & & & & \\
 & & & & & a_7 & b_7 & c_7 & & & & & & & & \\
 & & & & & & a_8 & b_8 & c_8 & & & & & & & \\
 & & & & & & & a_9 & b_9 & c_9 & & & & & & \\
 & & & & & & & & a_{10} & b_{10} & c_{10} & & & & \\
 & & & & & & & & & a_{11} & b_{11} & c_{11} & & & & \\
 & & & & 0 & & & & & & a_{12} & b_{12} & c_{12} & & & \\
 & & & & & & & & & & & a_{13} & b_{13} & c_{13} & & \\
 & & & & & & & & & & & & a_{14} & b_{14} & c_{14} & \\
 & & & & & & & & & & & & & a_{15} & b_{15} & c_{15} \\
 & & & & & & & & & & & & & & a_{16} & b_{16}
\end{pmatrix}$$

We then have n=16, m=8, q=$\log_2$n. The matrix A can be rewritten in the partitioned form given in (5.3).

In other words, we will rewrite the coefficient matrix A as the sum of a block diagonal matrix J with the summation of the products

$$\mathbf{x}^{(1)}\,\mathbf{y}^{(1)T}, \quad \mathbf{x}^{(2)}\,\mathbf{y}^{(2)T}, \quad \mathbf{x}^{(3)}\,\mathbf{y}^{(3)T}, \quad \mathbf{x}^{(4)}\,\mathbf{y}^{(4)T},$$
$$\mathbf{x}^{(5)}\,\mathbf{y}^{(5)T}, \quad \mathbf{x}^{(6)}\,\mathbf{y}^{(6)T}, \quad \mathbf{x}^{(7)}\,\mathbf{y}^{(7)T}$$

Vectors $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$ are all of dimension n=16 and of the type described in (5.5) and (5.6) respectively.

$$
A = \begin{pmatrix}
e_1 & c_1 & & & & & & & & & & & & & & \\
a_2 & e_2 & & & & & & & & & & & & & & \\
 & & e_3 & c_3 & & & & & & & & & & & & \\
 & & a_4 & e_4 & & & & & & & & & & & & \\
 & & & & e_5 & c_5 & & & & & & 0 & & & & \\
 & & & & a_6 & e_6 & & & & & & & & & & \\
 & & & & & & e_7 & c_7 & & & & & & & & \\
 & & & & & & a_8 & e_8 & & & & & & & & \\
 & & & & & & & & e_9 & c_9 & & & & & & \\
 & & & & & & & & a_{10} & e_{10} & & & & & & \\
 & & & & & & & & & & e_{11} & c_{11} & & & & \\
 & & & & 0 & & & & & & a_{12} & e_{12} & & & & \\
 & & & & & & & & & & & & e_{13} & c_{13} & & \\
 & & & & & & & & & & & & a_{14} & e_{14} & & \\
 & & & & & & & & & & & & & & e_{15} & c_{15} \\
 & & & & & & & & & & & & & & a_{16} & e_{16}
\end{pmatrix}
$$

$$
+ \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ a_3 \\ c_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ a_5 \\ c_4 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ a_7 \\ c_6 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ a_9 \\ c_8 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T}
$$

$$
+ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ a_{11} \\ c_{10} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ a_{13} \\ c_{12} \\ 0 \\ 0 \\ 0 \end{pmatrix}^{(j)T} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ a_{15} \\ c_{14} \\ 0 \end{pmatrix}^{(j)T}
$$

The new elements in matrix J are given by

$$e_1 = b_1\ , \qquad e_2 = b_2 - a_3\ , \qquad e_3 = b_3 - c_2\ , \qquad e_4 = b_4 - a_5\ ,$$

$$e_5 = b_5 - c_4\ , \qquad e_6 = b_6 - a_7\ , \qquad e_7 = b_7 - c_6\ , \qquad e_8 = b_8 - a_9\ ,$$

$$e_9 = b_9 - c_8\ , \qquad e_{10} = b_{10} - a_{11}\ , \qquad e_{11} = b_{11} - c_{10}\ , \qquad e_{12} = b_{12} - a_{13}\ ,$$

$$e_{13} = b_{13} - c_{12}\ , \qquad e_{14} = b_{14} - a_{15}\ , \qquad e_{15} = b_{15} - c_{14}\ , \qquad e_{16} = b_{16}.$$

The block diagonal matrix J is formed by $m = 8$ blocks, of which each one is a 2x2 matrix and therefore is directly invertible, i.e.

$$J_1 = \begin{pmatrix} e_1 & c_1 \\ a_2 & e_2 \end{pmatrix}, \quad J_1^{-1} = \Delta_1^{-1} \begin{pmatrix} e_2 & -c_1 \\ -a_2 & e_1 \end{pmatrix} \quad \text{where } \Delta_1 = (e_2 e_1 - a_2 c_1)$$

$$J_2 = \begin{pmatrix} e_3 & c_3 \\ a_4 & e_4 \end{pmatrix}, \quad J_2^{-1} = \Delta_2^{-1} \begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix} \quad \text{where } \Delta_2 = (e_4 e_3 - a_4 c_3)$$

We continue this process for all the remaining matrices up to and including J_8.

Let us now partition vectors $\mathbf{u}$ and $\mathbf{d}$ as required in (5.14):

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \\ u_{13} \\ u_{14} \\ u_{15} \\ u_{16} \end{pmatrix} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \\ \mathbf{u}_5 \\ \mathbf{u}_6 \\ \mathbf{u}_7 \\ \mathbf{u}_8 \end{pmatrix} \qquad \mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \\ d_{11} \\ d_{12} \\ d_{13} \\ d_{14} \\ d_{15} \\ d_{16} \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \\ \mathbf{d}_4 \\ \mathbf{d}_5 \\ \mathbf{d}_6 \\ \mathbf{d}_7 \\ \mathbf{d}_8 \end{pmatrix}$$

We have now completed the Preliminary Stage and are ready to start Stage 1, that is solve 8 subsystems of the form

$$J_i \mathbf{u}_i = \mathbf{d}_i \qquad \text{as } i \text{ ranges from 1 to 8.}$$

We obtain

$$\mathbf{u}_1 = J_1^{-1}\mathbf{d}_1 = \frac{1}{\Delta_1}\begin{pmatrix} e_2 & -c_1 \\ -a_2 & e_1 \end{pmatrix}\mathbf{d}_1$$

$$= \frac{1}{(e_1e_2 - a_2c_1)}\begin{pmatrix} e_2 & -c_1 \\ -a_2 & e_1 \end{pmatrix}\begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

$$= \frac{1}{(e_1e_2 - a_2c_1)}\begin{pmatrix} e_2d_1 - c_1d_2 \\ -a_2d_1 + e_1d_2 \end{pmatrix}$$

$$\mathbf{u}_2 = J_2^{-1}\mathbf{d}_2 = \frac{1}{\Delta_2}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\mathbf{d}_2$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\begin{pmatrix} d_3 \\ d_4 \end{pmatrix}$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} e_4d_3 - c_3d_4 \\ -a_4d_3 + e_3d_4 \end{pmatrix}$$

Analogous calculations are to be performed in order to obtain $\mathbf{u}_3$, $\mathbf{u}_4$, $\mathbf{u}_5$, $\mathbf{u}_6$, $\mathbf{u}_7$, $\mathbf{u}_8$. We have then completed Stage 1, obtaining the vector $\mathbf{u}$.

The subsystems in Stage 2, i.e.

$$J_j \mathbf{g}_i^{(j)} = \mathbf{x}_i^{(j)} \qquad \text{with} \quad i = 1, 2, .., 8 \quad \text{and} \quad j = 1, 2, 3$$

can also be evaluated explicitly.

The solutions are

$$\mathbf{g}_1^{(1)} = J_1^{-1}\mathbf{x}_1^{(1)} = \frac{1}{\Delta_1}\begin{pmatrix} e_2 & -c_1 \\ -a_2 & e_1 \end{pmatrix}\mathbf{x}_1^{(1)}$$

$$= \frac{1}{(e_1e_2 - a_2c_1)}\begin{pmatrix} e_2 & -c_1 \\ -a_2 & e_1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$= \frac{1}{(e_1e_2 - a_2c_1)}\begin{pmatrix} -c_1 \\ e_1 \end{pmatrix}$$

$$\mathbf{g}_2^{(1)} = J_2^{-1}\mathbf{x}_2^{(1)} = \frac{1}{\Delta_2}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\mathbf{x}_2^{(1)}$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} e_4 \\ -a_4 \end{pmatrix}$$

There is no need to calculate $\mathbf{g}_3^{(1)}$, $\mathbf{g}_4^{(1)}$, $\mathbf{g}_5^{(1)}$, $\mathbf{g}_6^{(1)}$, $\mathbf{g}_7^{(1)}$, $\mathbf{g}_8^{(1)}$, since their resulting components are clearly all equal to zero.

Then

$$\mathbf{g}^{(1)} = \begin{pmatrix} \mathbf{g}_1^{(1)} \\ \mathbf{g}_2^{(1)} \\ \mathbf{g}_3^{(1)} \\ \mathbf{g}_4^{(1)} \\ \mathbf{g}_5^{(1)} \\ \mathbf{g}_6^{(1)} \\ \mathbf{g}_7^{(1)} \\ \mathbf{g}_8^{(1)} \end{pmatrix} = \begin{pmatrix} -c_1/\Delta_1 \\ e_1/\Delta_1 \\ e_4/\Delta_2 \\ -a_4/\Delta_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The same observations hold for $\mathbf{g}^{(2)}$, i.e.

$$\mathbf{g}_2^{(2)} = J_2^{-1}\mathbf{x}_2^{(2)} = \frac{1}{\Delta_2}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\mathbf{x}_2^{(2)}$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} e_4 & -c_3 \\ -a_4 & e_3 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$= \frac{1}{(e_3e_4 - a_4c_3)}\begin{pmatrix} -c_3 \\ e_3 \end{pmatrix}$$

$$\mathbf{g}_3^{(2)} = J_3^{-1}\mathbf{x}_3^{(2)} = \frac{1}{\Delta_3}\begin{pmatrix} e_6 & -c_5 \\ -a_6 & e_5 \end{pmatrix}\mathbf{x}_3^{(2)}$$

$$= \frac{1}{(e_5e_6 - a_6c_5)}\begin{pmatrix} e_6 & -c_5 \\ -a_6 & e_5 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$= \frac{1}{(e_5e_6 - a_6c_5)}\begin{pmatrix} e_6 \\ -a_6 \end{pmatrix}$$

As before, there is no need to calculate $\mathbf{g}_1^{(2)}$, $\mathbf{g}_4^{(2)}$, $\mathbf{g}_5^{(2)}$, $\mathbf{g}_6^{(2)}$, $\mathbf{g}_7^{(2)}$, $\mathbf{g}_8^{(2)}$, since their components are equal to zero.

Then we obtain

$$\mathbf{g}^{(2)} = \begin{pmatrix} \mathbf{g}_1^{(2)} \\ \mathbf{g}_2^{(2)} \\ \mathbf{g}_3^{(2)} \\ \mathbf{g}_4^{(2)} \\ \mathbf{g}_5^{(2)} \\ \mathbf{g}_6^{(2)} \\ \mathbf{g}_7^{(2)} \\ \mathbf{g}_8^{(2)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -c_3/\Delta_2 \\ e_3/\Delta_2 \\ e_6/\Delta_3 \\ -a_6/\Delta_3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

With a similar process $\mathbf{g}^{(3)}$, $\mathbf{g}^{(4)}$, $\mathbf{g}^{(5)}$, $\mathbf{g}^{(6)}$, $\mathbf{g}^{(7)}$, $\mathbf{g}^{(8)}$ are calculated. This completes Stage 2.

Finally, the third stage of the algorithm gives the required solution $\mathbf{u}$, after a recursive series of updating operations. More specifically, during the first iteration of Stage 3, vector $\mathbf{u}$ is updated by applying the Sherman-Morrison formula to successive subsets of its components; this is done by using the corresponding components of vectors $\mathbf{g}^{(1)}$, $\mathbf{g}^{(3)}$, $\mathbf{g}^{(5)}$, $\mathbf{g}^{(7)}$, $\mathbf{y}^{(1)}$, $\mathbf{y}^{(3)}$, $\mathbf{y}^{(5)}$, $\mathbf{y}^{(7)}$. We can represent this process as follows

$$\mathbf{u} \begin{cases} \mathbf{u}_{1-2} = \left(I - \alpha_1 \mathbf{g}_{1-2}^{(1)} \mathbf{y}_{1-2}^{(1)T}\right) \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} \\ \mathbf{u}_{3-4} = \left(I - \alpha_3 \mathbf{g}_{3-4}^{(3)} \mathbf{y}_{3-4}^{(3)T}\right) \begin{pmatrix} \mathbf{u}_3 \\ \mathbf{u}_4 \end{pmatrix} \\ \mathbf{u}_{5-6} = \left(I - \alpha_5 \mathbf{g}_{5-6}^{(5)} \mathbf{y}_{5-6}^{(5)T}\right) \begin{pmatrix} \mathbf{u}_5 \\ \mathbf{u}_6 \end{pmatrix} \\ \mathbf{u}_{7-8} = \left(I - \alpha_7 \mathbf{g}_{7-8}^{(7)} \mathbf{y}_{7-8}^{(7)T}\right) \begin{pmatrix} \mathbf{u}_7 \\ \mathbf{u}_8 \end{pmatrix} \end{cases}$$

The graph above can be read in the following way:

$$\mathbf{u} \begin{cases} \text{components from 1 to 4 of vector } \mathbf{u} \text{ are updated} \\ \text{components from 5 to 8 of vector } \mathbf{u} \text{ are updated} \\ \text{components from 9 to 12 of vector } \mathbf{u} \text{ are updated} \\ \text{components from 13 to 16 of vector } \mathbf{u} \text{ are updated} \end{cases}$$

Vectors $\mathbf{g}^{(2)}$, $\mathbf{g}^{(4)}$, $\mathbf{g}^{(6)}$ are also updated by an identical process, that is to say by using the corresponding components of $\mathbf{g}^{(1)}$, $\mathbf{g}^{(3)}$, $\mathbf{g}^{(5)}$, $\mathbf{g}^{(7)}$.

To first update vector $\mathbf{g}^{(2)}$, for example, we calculate:

$$\mathbf{g}^{(2)} \begin{cases} \mathbf{g}_{1-2}^{(2)} = \left(I - \alpha_1 \mathbf{g}_{1-2}^{(1)} \mathbf{y}_{1-2}^{(1)T}\right) \mathbf{g}_{1-2}^{(2)} \\ \mathbf{g}_{3-4}^{(2)} = \left(I - \alpha_3 \mathbf{g}_{3-4}^{(3)} \mathbf{y}_{3-4}^{(3)T}\right) \mathbf{g}_{3-4}^{(2)} \\ \mathbf{g}_{5-6}^{(2)} = \left(I - \alpha_5 \mathbf{g}_{5-6}^{(5)} \mathbf{y}_{5-6}^{(5)T}\right) \mathbf{g}_{5-6}^{(2)} \\ \mathbf{g}_{7-8}^{(2)} = \left(I - \alpha_7 \mathbf{g}_{7-8}^{(7)} \mathbf{y}_{7-8}^{(7)T}\right) \mathbf{g}_{7-8}^{(2)} \end{cases}$$

that means

$$\mathbf{g}^{(2)} \begin{cases} \text{components from 1 to 4 of vector } \mathbf{g}^{(2)} \text{ are updated} \\ \text{components from 5 to 8 of vector } \mathbf{g}^{(2)} \text{ are updated} \\ \text{components from 9 to 12 of vector } \mathbf{g}^{(2)} \text{ are updated} \\ \text{components from 13 to 16 of vector } \mathbf{g}^{(2)} \text{ are updated} \end{cases}$$

We complete the first iteration by calculating the updated values of vectors $\mathbf{g}^{(4)}$ and $\mathbf{g}^{(6)}$ in an analogous way as $\mathbf{g}^{(2)}$. Then, we again update vector $\mathbf{u}$, starting the second iteration of the recursive process. This time, $\mathbf{u}$ is updated by using the corresponding components of vectors $\mathbf{g}^{(2)}$ and $\mathbf{g}^{(6)}$, along with $\mathbf{y}^{(2)}$ and $\mathbf{y}^{(6)}$:

$$\mathbf{u} \begin{cases} \mathbf{u}_{1-4} = \left(I - \alpha_2 \mathbf{g}_{1-4}^{(2)} \mathbf{y}_{1-4}^{(2)T}\right) \begin{pmatrix} \mathbf{u}_{1-2} \\ \mathbf{u}_{3-4} \end{pmatrix} \\ \mathbf{u}_{5-8} = \left(I - \alpha_6 \mathbf{g}_{5-8}^{(6)} \mathbf{y}_{5-8}^{(6)T}\right) \begin{pmatrix} \mathbf{u}_{5-6} \\ \mathbf{u}_{7-8} \end{pmatrix} \end{cases}$$

As before, the graph above can be read in the following way:

$$\mathbf{u} \begin{cases} \text{components from 1 to 8 of vector } \mathbf{u} \text{ are updated} \\ \text{components from 9 to 16 of vector } \mathbf{u} \text{ are updated} \end{cases}$$

The vector $\mathbf{g}^{(4)}$ is also updated by using the corresponding components of $\mathbf{g}^{(2)}$ and $\mathbf{g}^{(6)}$:

$$\mathbf{g}^{(4)} \begin{cases} \mathbf{g}_{1-4}^{(4)} = \left(I - \alpha_2 \mathbf{g}_{1-4}^{(2)} \mathbf{y}_{1-4}^{(2)T}\right) \mathbf{g}_{1-4}^{(4)} \\ \mathbf{g}_{5-8}^{(4)} = \left(I - \alpha_6 \mathbf{g}_{5-8}^{(6)} \mathbf{y}_{5-8}^{(6)T}\right) \mathbf{g}_{5-8}^{(4)} \end{cases}$$

that means

$$\mathbf{g}^{(4)} \begin{cases} \text{components from 1 to 8 of vector } \mathbf{g}^{(4)} \text{ are updated} \\ \text{components from 9 to 16 of vector } \mathbf{g}^{(4)} \text{ are updated} \end{cases}$$

This ends the second iteration.

By performing the third iteration, we finally obtain the solution vector $\mathbf{u}$; this time, we update all the 16 components of $\mathbf{u}$ by using the corresponding components of vector $\mathbf{g}^{(4)}$:

$$\mathbf{u} = \mathbf{u}_{1-8} = \left(I - \alpha_4 \mathbf{g}_{1-8}^{(4)} \mathbf{y}_{1-8}^{(4)T}\right) \begin{pmatrix} \mathbf{u}_{1-4} \\ \mathbf{u}_{5-8} \end{pmatrix}$$

$$= \left(I - \alpha_4 \mathbf{g}^{(4)} \mathbf{y}^{(4)T}\right) \begin{pmatrix} \mathbf{u}_{1-4} \\ \mathbf{u}_{5-8} \end{pmatrix}$$

Note that in this example we have required $q - 1 = 3$ iterations to complete Stage 3 and to obtain the final solution vector $\mathbf{u}$.

5.6. A Numerical Example

The Recursive Decoupling procedure can be further illustrated by the following simple numerical example [14], involving the 8x8 linear system

$$\begin{pmatrix} 2 & -1 & & & & & & \\ -1 & 2 & -1 & & & & 0 & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & -1 & 2 & -1 & & \\ & & & & -1 & 2 & -1 & \\ & & 0 & & & -1 & 2 & -1 \\ & & & & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The system matrix can be decoupled into the form

$$J + \mathbf{x}^{(1)}\mathbf{y}^{(1)T} + \mathbf{x}^{(2)}\mathbf{y}^{(2)T} + \mathbf{x}^{(3)}\mathbf{y}^{(3)T}$$

as follows

$$\begin{pmatrix} 2 & -1 & & & & & & \\ -1 & 3 & & & & & 0 & \\ & & 3 & -1 & & & & \\ & & -1 & 3 & & & & \\ & & & & 3 & -1 & & \\ & & & & -1 & 3 & & \\ & & 0 & & & & 3 & -1 \\ & & & & & & -1 & 2 \end{pmatrix}$$

$$+ \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \end{pmatrix}^T$$

This concludes our Pre-stage.

We continue with an illustration of the following three stages.

Stage 1. Find the solution to the system $J\mathbf{u} = \mathbf{d}$, that is, solve the 4 subsystems:

$$\mathbf{u}_1 = J_1^{-1}\mathbf{d}_1 = \frac{1}{\Delta_1}\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \frac{1}{(6-1)}\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3/5 \\ 1/5 \end{pmatrix}$$

$$\mathbf{u}_2 = J_2^{-1}\mathbf{d}_2 = \frac{1}{\Delta_2}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} d_3 \\ d_4 \end{pmatrix} = \frac{1}{(9-1)}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathbf{u}_3 = J_3^{-1}\mathbf{d}_3 = \frac{1}{\Delta_3}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} d_5 \\ d_6 \end{pmatrix} = \frac{1}{(9-1)}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathbf{u}_4 = J_4^{-1}\mathbf{d}_4 = \frac{1}{\Delta_4}\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} d_7 \\ d_8 \end{pmatrix} = \frac{1}{(6-1)}\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1/5 \\ 3/5 \end{pmatrix}$$

Therefore $\mathbf{u}$, at the end of stage 1, is given by

$$\mathbf{u} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \end{pmatrix} = \begin{pmatrix} 3/5 \\ 1/5 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1/5 \\ 3/5 \end{pmatrix}$$

Stage 2. Find the solution to the systems

$$J\,\mathbf{g}^{(1)} = \mathbf{x}^{(1)}$$

$$J\,\mathbf{g}^{(2)} = \mathbf{x}^{(2)}$$

$$J\,\mathbf{g}^{(3)} = \mathbf{x}^{(3)}$$

.

For each of the three systems, this is equivalent to solving 4 subsystems

$$\mathbf{g}_1^{(1)} = J_1^{-1}\mathbf{x}_1^{(1)} = \frac{1}{\Delta_1}\begin{pmatrix}3 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \begin{pmatrix}1/5\\ 2/5\end{pmatrix}$$

$$\mathbf{g}_2^{(1)} = J_1^{-1}\mathbf{x}_2^{(1)} = \frac{1}{\Delta_2}\begin{pmatrix}3 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}3/8\\ 1/8\end{pmatrix}$$

We do not calculate $\mathbf{g}_3^{(1)}$ and $\mathbf{g}_4^{(1)}$, since their components are zero.

$$\mathbf{g}_2^{(2)} = J_2^{-1}\mathbf{x}_2^{(2)} = \frac{1}{\Delta_2}\begin{pmatrix}3 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \begin{pmatrix}1/8\\ 3/8\end{pmatrix}$$

$$\mathbf{g}_3^{(2)} = J_3^{-1}\mathbf{x}_3^{(2)} = \frac{1}{\Delta_3}\begin{pmatrix}3 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}3/8\\ 1/8\end{pmatrix}$$

As before, we have not calculated some zero-valued components $\mathbf{g}_1^{(2)}$ and $\mathbf{g}_4^{(2)}$.

Components $\mathbf{g}_1^{(3)}$ and $\mathbf{g}_2^{(3)}$ are zero in the third system and we have

$$\mathbf{g}_3^{(3)} = J_3^{-1}\mathbf{x}_3^{(3)} = \frac{1}{\Delta_3}\begin{pmatrix}3 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \begin{pmatrix}1/8\\ 3/8\end{pmatrix}$$

$$\mathbf{g}_3^{(4)} = J_4^{-1}\mathbf{x}_4^{(3)} = \frac{1}{\Delta_4}\begin{pmatrix}2 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}2/5\\ 1/5\end{pmatrix}$$

At the end of stage 2, we have obtained:

$$\mathbf{g}^{(1)} = \begin{pmatrix}1/5\\ 2/5\\ 3/8\\ 1/8\\ 0\\ 0\\ 0\\ 0\end{pmatrix} \quad \mathbf{g}^{(2)} = \begin{pmatrix}0\\ 0\\ 1/8\\ 3/8\\ 3/8\\ 1/8\\ 0\\ 0\end{pmatrix} \quad \mathbf{g}^{(3)} = \begin{pmatrix}0\\ 0\\ 0\\ 0\\ 1/8\\ 3/8\\ 2/5\\ 1/5\end{pmatrix}$$

Stage 3. The updating stage can now be carried out:

$$\alpha_1 = \frac{1}{\left(1 + \mathbf{y}^{(1)T}\,\mathbf{g}^{(1)}\right)} = \frac{1}{\left(1 + \sum_{i=1}^{8} \mathbf{y}_i^{(1)T}\,\mathbf{g}_i^{(1)}\right)}$$

$$= \frac{1}{\left(1 + (-1)\frac{2}{5} + (-1)\frac{3}{8}\right)} = \frac{40}{9}$$

$$\mathbf{g}^{(1)}\,\mathbf{y}^{(1)T} = \begin{pmatrix} 0 & -1/5 & -1/5 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2/5 & -2/5 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3/8 & -3/8 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1/8 & -1/8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\alpha_1\,\mathbf{g}^{(1)}\,\mathbf{y}^{(1)T} = \begin{pmatrix} 0 & -8/9 & -8/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & -16/9 & -16/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & -15/9 & -15/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & -5/9 & -5/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$I - \alpha_1\,\mathbf{g}^{(1)}\,\mathbf{y}^{(1)T} = \begin{pmatrix} 1 & 8/9 & 8/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 25/9 & 16/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 15/9 & 24/9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5/9 & 5/9 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{u}_{1-2} = \left(I - \alpha_1\,\mathbf{g}_{1-2}^{(1)}\,\mathbf{y}_{1-2}^{(1)T}\right)\begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 8/9 & 8/9 & 0 \\ 0 & 25/9 & 16/9 & 0 \\ 0 & 15/9 & 24/9 & 0 \\ 0 & 5/9 & 5/9 & 0 \end{pmatrix}\begin{pmatrix} 3/5 \\ 1/5 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 7/9 \\ 5/9 \\ 3/9 \\ 1/9 \end{pmatrix}$$

$$\alpha_3 = \frac{1}{\left(1 + \mathbf{y}^{(3)T}\,\mathbf{g}^{(3)}\right)} = \cdots = \frac{1}{\left(1 - \frac{3}{8} - \frac{2}{5}\right)} = \frac{40}{9}$$

$$\mathbf{g}^{(3)}\,\mathbf{y}^{(3)T} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/8 & -1/8 & 0 \\ 0 & 0 & 0 & 0 & 0 & -3/8 & -3/8 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2/5 & -2/5 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/5 & -1/5 & 0 \end{pmatrix}$$

$$\alpha_3\,\mathbf{g}^{(3)}\,\mathbf{y}^{(3)T} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -5/9 & -5/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & -15/9 & -15/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & -16/9 & -16/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & -8/9 & -8/9 & 0 \end{pmatrix}$$

$$I - \alpha_3\,\mathbf{g}^{(3)}\,\mathbf{y}^{(3)T} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 5/9 & 5/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 24/9 & 15/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 16/9 & 25/9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8/9 & 8/9 & 1 \end{pmatrix}$$

$$\mathbf{u}_{3-4} = \left(I - \alpha_3\,\mathbf{g}^{(3)}_{3-4}\,\mathbf{y}^{(3)T}_{3-4}\right)\begin{pmatrix} \mathbf{u}_3 \\ \mathbf{u}_4 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 5/9 & 5/9 & 0 \\ 0 & 24/9 & 15/9 & 0 \\ 0 & 16/9 & 25/9 & 0 \\ 0 & 8/9 & 8/9 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1/5 \\ 3/5 \end{pmatrix} = \begin{pmatrix} 1/9 \\ 3/9 \\ 5/9 \\ 7/9 \end{pmatrix}$$

The updated vector $\mathbf{u}$ is then:

$$\mathbf{u} = \begin{pmatrix} 7/9 \\ 5/9 \\ 3/9 \\ 1/9 \\ 1/9 \\ 3/9 \\ 5/9 \\ 7/9 \end{pmatrix}$$

To complete iteration 1 of stage 3, we still have to update vector $\mathbf{g}^{(2)}$:

$$\mathbf{g}_{1-2}^{(2)} = \left(I - \alpha_1 \, \mathbf{g}_{1-2}^{(1)} \, \mathbf{y}_{1-2}^{(1)T}\right)$$

$$= \begin{pmatrix} 1 & 8/9 & 8/9 & 0 \\ 0 & 25/9 & 16/9 & 0 \\ 0 & 15/9 & 24/9 & 0 \\ 0 & 5/9 & 5/9 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1/8 \\ 3/8 \end{pmatrix} = \begin{pmatrix} 1/9 \\ 2/9 \\ 3/9 \\ 4/9 \end{pmatrix}$$

$$\mathbf{g}_{3-4}^{(2)} = \left(I - \alpha_3 \, \mathbf{g}_{3-4}^{(3)} \, \mathbf{y}_{3-4}^{(3)T}\right)$$

$$= \begin{pmatrix} 1 & 5/9 & 5/9 & 0 \\ 0 & 24/9 & 15/9 & 0 \\ 0 & 16/9 & 25/9 & 0 \\ 0 & 8/9 & 8/9 & 1 \end{pmatrix} \begin{pmatrix} 3/8 \\ 1/8 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 4/9 \\ 3/9 \\ 2/9 \\ 1/9 \end{pmatrix}$$

The updated vector $\mathbf{g}^{(2)}$ is then:

$$\mathbf{g}^{(2)} = \begin{pmatrix} 1/9 \\ 2/9 \\ 3/9 \\ 4/9 \\ 4/9 \\ 3/9 \\ 2/9 \\ 1/9 \end{pmatrix}$$

This completes iteration 1.

Iteration 2 of stage 3 consists of updating vectors $\mathbf{u}$ by means of $\mathbf{g}^{(2)}$.

$$\alpha_2 = \frac{1}{\left(1 + \mathbf{y}^{(2)T}\,\mathbf{g}^{(2)}\right)} = \cdots = \frac{1}{\left(1 - \frac{4}{9} - \frac{4}{9}\right)} = 9$$

$$\mathbf{g}^{(2)}\,\mathbf{y}^{(2)T} = \begin{pmatrix} 0 & 0 & 0 & -1/9 & -1/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2/9 & -2/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3/9 & -3/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4/9 & -4/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4/9 & -4/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3/9 & -3/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2/9 & -2/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/9 & -1/9 & 0 & 0 & 0 \end{pmatrix}$$

$$\alpha_2\,\mathbf{g}^{(2)}\,\mathbf{y}^{(2)T} = \begin{pmatrix} 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \end{pmatrix}$$

$$I - \alpha_2\,\mathbf{g}^{(2)}\,\mathbf{y}^{(2)T} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{u} = \mathbf{u}_{1-4} = \left(I - \alpha_2\,\mathbf{g}^{(2)}_{1-4}\,\mathbf{y}^{(2)T}_{1-4}\right)\begin{pmatrix} \mathbf{u}_{1-2} \\ \mathbf{u}_{3-4} \end{pmatrix}$$

$$= \left(I - \alpha_2\,\mathbf{g}^{(2)}\,\mathbf{y}^{(2)T}\right)\begin{pmatrix} 7/9 \\ 5/9 \\ 3/9 \\ 1/9 \\ 1/9 \\ 3/9 \\ 5/9 \\ 7/9 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

This is the final solution.

Note that we required $q - 1 = 2$ iterations during stage 3, in order to obtain the solution vector $\mathbf{u}$.

5.7. The Recursive Decoupling Routine

In this section we will describe the Fortran routine implementing the Recursive Decoupling Algorithm [12, 25].

Since our coefficient matrix A is a sparse matrix of tridiagonal form, we have used three n-dimensional vectors to store its elements, which are given by:

a which is used to store the sub-diagonal values;

b which is used to store the main diagonal values;

c which is used to store the upper diagonal values.

The arrays **u** and **d**, both of dimension n, are used to store the unknown and known vectors. As the routine is completed, the calculated solution vector is stored in **u**.

The partitioning matrix J is also a sparse matrix of the form given in formula (5.3) (see also the matrix notation given in (5.8)). To initialise the matrix J we need only to introduce one further n-dimensional vector **e**, whose values are defined by expressions (5.4). Note that there is no need to obtain an explicit initialisation of the matrix J: we are only interested in finding the inverses J_i^{-1} of each 2x2 block matrix J_i to form the inverse of the partitioning matrix itself. This is done by applying formulae (5.13),

i.e as index i ranges from 1 to m

we calculate each real number Δ_i;

then we store its inverse value in a work variable called *rec*;

then we initialise each 2x2 matrix J_i^{-1} using (5.13)

In order to retain a parallel structure during the algorithmic stages, we have grouped all the matrices J_i^{-1} into a single array *JJ1* of dimension 2x2xm. The last index i of the array *JJ1* ranges from 1 to m and gives the current submatrix J_i^{-1}.

To complete the Pre-Stage section we still have to define a memory structure to retain vectors $\mathbf{x}^{(j)}$, $\mathbf{y}^{(j)}$ and $\mathbf{g}^{(j)}$. In an analogous way to array *JJ1*, we have chosen to store these n-dimensional vectors in three different matrices, each one of dimension $nx(q-1)$. The vectors $\mathbf{x}^{(j)}$ are stored in the first of these $nx(q-1)$ matrices, the vectors $\mathbf{y}^{(j)}$ are stored into the second matrix and the third $nx(q-1)$ matrix is used to store the vectors $\mathbf{g}^{(j)}$. As in the array *JJ1*, the second index j of each matrix ranges from 1 to $q-1$ and gives the current vector $\mathbf{x}^{(j)}$ (or $\mathbf{y}^{(j)}$, or $\mathbf{g}^{(j)}$, depending upon which of the three matrices we are considering).

To explain why the three matrices have been used in this way and to explain why the second index j only varies from 1 to $q-1$ in particular (instead of varying from 1 to $m-1$ as we would expect), we need to make some observations.

The main problems we have met in implementing this routine were related to maintaining the intrinsic parallel nature of the Recursive Decoupling Algorithm. We had to study a storage structure that was best suited to performing Stage 1, Stage 2 and Stage 3 in parallel .

Fortran was the chosen programming language, that does not allow the explicit use of any kind of tree structure, unlike other computer languages such as Pascal or C. Particular care is needed in the implementation of the Fan-in graph shown in figure (5.17) and the main problem, therefore, consisted of finding some way to simulate the tree graph.

Storing all the vectors $\mathbf{x}^{(j)}$ in an $nx(q-1)$ matrix X and all the vectors $\mathbf{g}^{(j)}$ in an $nx(q-1)$ matrix G appeared to be the best way to simulate the Fan-in figure. We will show later on that it is possible to obtain a saving of memory allocation, by using the same $nx(q-1)$ matrix X to overwrite the values of vectors $\mathbf{y}^{(j)}$.

Let us justify the above choice.

The tree structure (5.17) is made of $q - 1 = \log_2 m$ levels. At each level, only a few components of the vectors $\mathbf{u}$, $\mathbf{x}^{(j)}$, $\mathbf{y}^{(j)}$ and $\mathbf{g}^{(j)}$ are involved in the calculations (refer again to figure (5.17)). Let us follow what happens to these vectors during the three stages of the algorithm, and in particular during Stage 3.

As we have already mentioned, in Stage 1 we obtain the vector $\mathbf{u}$ by partitioning $\mathbf{u}$ and $\mathbf{d}$ according to (5.14) and solving the m subsystems (5.19). At the end of this first stage, vector $\mathbf{u}$ then contains n values, giving an initial approximate solution to our original problem. This initial value is given by the solution of $J\mathbf{u} = \mathbf{d}$.

In Stage 2, we obtain each vector $\mathbf{g}^{(j)}$, after defining the corresponding vector $\mathbf{x}^{(j)}$ as in (5.5) and solving the m subsystems (5.22). This is done for each of the $m - 1$ vectors $\mathbf{g}^{(j)}$. At the end of the second stage, however, each $\mathbf{g}^{(j)}$ contains only 2 non-zero elements (in the $(2j)^{th}$ and the $(2j+1)^{th}$ positions).

It might now be possible to store all the non-zero values of the vectors $\mathbf{g}^{(j)}$, in one array $\mathbf{G}^{(1)}$ of dimension n as follows

$$\mathbf{G}^{(1)} = \begin{pmatrix} G_1^{(1)} \\ G_2^{(1)} \\ G_3^{(1)} \\ G_4^{(1)} \\ G_5^{(1)} \\ G_6^{(1)} \\ G_7^{(1)} \\ G_8^{(1)} \\ \vdots \\ G_{n-3}^{(1)} \\ G_{n-2}^{(1)} \\ G_{n-1}^{(1)} \\ G_n^{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_2^{(1)} \\ \mathbf{g}_3^{(1)} \\ \mathbf{g}_4^{(2)} \\ \mathbf{g}_5^{(2)} \\ \mathbf{g}_6^{(3)} \\ \mathbf{g}_7^{(3)} \\ \mathbf{g}_8^{(4)} \\ \mathbf{g}_9^{(4)} \\ \vdots \\ \mathbf{g}_{n-4}^{(m-2)} \\ \mathbf{g}_{n-3}^{(m-2)} \\ \mathbf{g}_{n-2}^{(m-1)} \\ \mathbf{g}_{n-1}^{(m-1)} \end{pmatrix}$$

Better still, in order to maintain the correspondence between suffixes, we could have stored all the non-zero values of vectors $\mathbf{g}^{(j)}$ in two arrays $\mathbf{G}^{(1)}$, $\mathbf{G}^{(2)}$ of dimension n as follows

$$\mathbf{G}^{(1)} = \begin{pmatrix} G_1^{(1)} \\ G_2^{(1)} \\ G_3^{(1)} \\ G_4^{(1)} \\ G_5^{(1)} \\ G_6^{(1)} \\ G_7^{(1)} \\ G_8^{(1)} \\ \vdots \\ G_{n-3}^{(1)} \\ G_{n-2}^{(1)} \\ G_{n-1}^{(1)} \\ G_n^{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_1^{(1)} \\ \mathbf{g}_2^{(1)} \\ \mathbf{g}_3^{(1)} \\ \mathbf{g}_4^{(1)} \\ \mathbf{g}_5^{(3)} \\ \mathbf{g}_6^{(3)} \\ \mathbf{g}_7^{(3)} \\ \mathbf{g}_8^{(3)} \\ \vdots \\ \mathbf{g}_{n-3}^{(m-1)} \\ \mathbf{g}_{n-2}^{(m-1)} \\ \mathbf{g}_{n-1}^{(m-1)} \\ \mathbf{g}_n^{(m-1)} \end{pmatrix} \qquad \mathbf{G}^{(2)} = \begin{pmatrix} G_1^{(2)} \\ G_2^{(2)} \\ G_3^{(2)} \\ G_4^{(2)} \\ G_5^{(2)} \\ G_6^{(2)} \\ G_7^{(2)} \\ G_8^{(2)} \\ \vdots \\ G_{n-3}^{(2)} \\ G_{n-2}^{(2)} \\ G_{n-1}^{(2)} \\ G_n^{(2)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{g}_3^{(2)} \\ \mathbf{g}_4^{(2)} \\ \mathbf{g}_5^{(2)} \\ \mathbf{g}_6^{(2)} \\ \mathbf{g}_7^{(4)} \\ \mathbf{g}_8^{(4)} \\ \vdots \\ \mathbf{g}_{n-3}^{(m-2)} \\ \mathbf{g}_{n-2}^{(m-2)} \\ 0 \\ 0 \end{pmatrix}$$

It turns out that neither of these methods permits us to maintain the parallel structure of the algorithm during the third stage.

Stage 3, in fact, is composed of $q-1$ iterations, corresponding to the $q-1$ levels of the tree structure (5.17). At each iteration, the number of non-zero components of vectors $\mathbf{g}^{(j)}$ varies; in fact the number of non-zero components doubles.

This only affects some of the vectors $\mathbf{g}^{(j)}$ and can be illustrated with the following diagram

$$\begin{pmatrix} g_1^{(1)} \\ g_2^{(1)} \\ g_3^{(1)} \\ g_4^{(1)} \end{pmatrix} \begin{pmatrix} g_3^{(2)} \\ g_4^{(2)} \\ g_5^{(2)} \\ g_6^{(2)} \end{pmatrix} \begin{pmatrix} g_5^{(3)} \\ g_6^{(3)} \\ g_7^{(3)} \\ g_8^{(3)} \end{pmatrix} \begin{pmatrix} g_7^{(4)} \\ g_8^{(4)} \\ g_9^{(4)} \\ g_{10}^{(4)} \end{pmatrix} \begin{pmatrix} g_9^{(5)} \\ g_{10}^{(5)} \\ g_{11}^{(5)} \\ g_{12}^{(5)} \end{pmatrix} \begin{pmatrix} g_{11}^{(6)} \\ g_{12}^{(6)} \\ g_{13}^{(6)} \\ g_{14}^{(6)} \end{pmatrix} \begin{pmatrix} g_{13}^{(7)} \\ g_{14}^{(7)} \\ g_{15}^{(7)} \\ g_{16}^{(7)} \end{pmatrix} \cdots$$

ITERATION 1

$$\begin{pmatrix} g_1^{(1)} \\ g_2^{(1)} \\ g_3^{(1)} \\ g_4^{(1)} \end{pmatrix} \begin{pmatrix} g_1^{(2)} \\ g_2^{(2)} \\ g_3^{(2)} \\ g_4^{(2)} \\ g_5^{(2)} \\ g_6^{(2)} \\ g_7^{(2)} \\ g_8^{(2)} \end{pmatrix} \begin{pmatrix} g_5^{(3)} \\ g_6^{(3)} \\ g_7^{(3)} \\ g_8^{(3)} \end{pmatrix} \begin{pmatrix} g_5^{(4)} \\ g_6^{(4)} \\ g_7^{(4)} \\ g_8^{(4)} \\ g_9^{(4)} \\ g_{10}^{(4)} \\ g_{11}^{(4)} \\ g_{12}^{(4)} \end{pmatrix} \begin{pmatrix} g_9^{(5)} \\ g_{10}^{(5)} \\ g_{11}^{(5)} \\ g_{12}^{(5)} \end{pmatrix} \begin{pmatrix} g_9^{(6)} \\ g_{10}^{(6)} \\ g_{11}^{(6)} \\ g_{12}^{(6)} \\ g_{13}^{(6)} \\ g_{14}^{(6)} \\ g_{15}^{(6)} \\ g_{16}^{(6)} \end{pmatrix} \begin{pmatrix} g_{13}^{(7)} \\ g_{14}^{(7)} \\ g_{15}^{(7)} \\ g_{16}^{(7)} \end{pmatrix} \cdots \quad (5.26)$$

ITERATION 2

$$\begin{pmatrix} g_1^{(1)} \\ g_2^{(1)} \\ g_3^{(1)} \\ g_4^{(1)} \end{pmatrix} \begin{pmatrix} g_1^{(2)} \\ g_2^{(2)} \\ g_3^{(2)} \\ g_4^{(2)} \\ g_5^{(2)} \\ g_6^{(2)} \\ g_7^{(2)} \\ g_8^{(2)} \end{pmatrix} \begin{pmatrix} g_5^{(3)} \\ g_6^{(3)} \\ g_7^{(3)} \\ g_8^{(3)} \end{pmatrix} \begin{pmatrix} g_1^{(4)} \\ g_2^{(4)} \\ g_3^{(4)} \\ g_4^{(4)} \\ g_5^{(4)} \\ g_6^{(4)} \\ g_7^{(4)} \\ g_8^{(4)} \\ g_9^{(4)} \\ g_{10}^{(4)} \\ g_{11}^{(4)} \\ g_{12}^{(4)} \\ g_{13}^{(4)} \\ g_{14}^{(4)} \\ g_{15}^{(4)} \\ g_{16}^{(4)} \end{pmatrix} \begin{pmatrix} g_9^{(5)} \\ g_{10}^{(5)} \\ g_{11}^{(5)} \\ g_{12}^{(5)} \end{pmatrix} \begin{pmatrix} g_9^{(6)} \\ g_{10}^{(6)} \\ g_{11}^{(6)} \\ g_{12}^{(6)} \\ g_{13}^{(6)} \\ g_{14}^{(6)} \\ g_{15}^{(6)} \\ g_{16}^{(6)} \end{pmatrix} \begin{pmatrix} g_{13}^{(7)} \\ g_{14}^{(7)} \\ g_{15}^{(7)} \\ g_{16}^{(7)} \end{pmatrix} \cdots$$

As a result of (5.26), we start iteration 1 of the third stage working on 4-dimensional vectors. In particular, vectors $\mathbf{g}^{(j)}$ with odd index j are used to update the remaining vectors $\mathbf{g}^{(j)}$ with j even. The vectors with even index double in size according to the above diagram.

We start iteration 2 working on 8-dimensional vectors. This time, the vectors $\mathbf{g}^{(j)}$ where $j = 2, 6, 10, ...$ are used to update vectors $\mathbf{g}^{(j)}$ with $j = 4, 8, 12, ...$ As before, these last vectors again double in size, and are now of dimension 16.

This process continues in a similar manner for each iteration step.

At the same time, the components of vector $\mathbf{u}$ are also updated as follows

$$[u_1 \quad \dots \quad u_4][u_5 \quad \dots \quad u_8][u_9 \quad \dots \quad u_{12}][u_{13} \quad \dots \quad u_{16}]\dots$$

ITERATION 1

$$[u_1 \quad \dots \quad u_8][u_9 \quad \dots \quad u_{16}]\dots \tag{5.27}$$

ITERATION 2

$$[u_1 \quad \dots \quad u_{16}]\dots$$

As a consequence of figures (5.26) and (5.27), we make the following observations:

- a single n-dimensional array is sufficient to store vector $\mathbf{u}$ throughout all the updating iterations, since the variations only affect the size of the subsets of components.
- in order to avoid the risk of overlapping between the old and the new components, we need more than the two vectors $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(2)}$ to store vectors $\mathbf{g}^{(j)}$.

To solve the component overlapping problem, let us consider $q - 1$ arrays of the same type as $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(2)}$. Each one has dimension n; they store the values of vectors $\mathbf{g}^{(j)}$ as shown in the following figure

$$\mathbf{G}^{(1)} = \begin{pmatrix} \mathbf{g}_1^{(1)} \\ \mathbf{g}_2^{(1)} \\ \mathbf{g}_3^{(1)} \\ \mathbf{g}_4^{(1)} \\ \mathbf{g}_5^{(3)} \\ \mathbf{g}_6^{(3)} \\ \mathbf{g}_7^{(3)} \\ \mathbf{g}_8^{(3)} \\ \mathbf{g}_9^{(5)} \\ \mathbf{g}_{10}^{(5)} \\ \mathbf{g}_{11}^{(5)} \\ \mathbf{g}_{12}^{(5)} \\ \mathbf{g}_{13}^{(7)} \\ \mathbf{g}_{14}^{(7)} \\ \mathbf{g}_{15}^{(7)} \\ \mathbf{g}_{16}^{(7)} \\ \vdots \end{pmatrix} \qquad \mathbf{G}^{(2)} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{g}_3^{(2)} \\ \mathbf{g}_4^{(2)} \\ \mathbf{g}_5^{(2)} \\ \mathbf{g}_6^{(2)} \\ 0 \\ 0 \\ 0 \\ 0 \\ \mathbf{g}_{11}^{(6)} \\ \mathbf{g}_{12}^{(6)} \\ \mathbf{g}_{13}^{(6)} \\ \mathbf{g}_{14}^{(6)} \\ 0 \\ 0 \\ \vdots \end{pmatrix} \qquad \mathbf{G}^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \mathbf{g}_7^{(4)} \\ \mathbf{g}_8^{(4)} \\ \mathbf{g}_9^{(4)} \\ \mathbf{g}_{10}^{(4)} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \ldots \quad (5.28)$$

Starting the first iteration, we will use $\mathbf{G}^{(1)}$ to update $\mathbf{G}^{(2)}$, $\mathbf{G}^{(3)}$,$\cdots$, $\mathbf{G}^{(q-1)}$. At the end of iteration 1, $\mathbf{G}^{(1)}$ remains unchanged, while the remaining vectors $\mathbf{G}^{(j)}$ are given by

$$\mathbf{G}^{(2)} = \begin{pmatrix} \mathbf{g}_1^{(2)} \\ \mathbf{g}_2^{(2)} \\ \mathbf{g}_3^{(2)} \\ \mathbf{g}_4^{(2)} \\ \mathbf{g}_5^{(2)} \\ \mathbf{g}_6^{(2)} \\ \mathbf{g}_7^{(2)} \\ \mathbf{g}_8^{(2)} \\ \mathbf{g}_9^{(6)} \\ \mathbf{g}_{10}^{(6)} \\ \mathbf{g}_{11}^{(6)} \\ \mathbf{g}_{12}^{(6)} \\ \mathbf{g}_{13}^{(6)} \\ \mathbf{g}_{14}^{(6)} \\ \mathbf{g}_{15}^{(6)} \\ \mathbf{g}_{16}^{(6)} \\ \vdots \end{pmatrix} \qquad \mathbf{G}^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \mathbf{g}_5^{(4)} \\ \mathbf{g}_6^{(4)} \\ \mathbf{g}_7^{(4)} \\ \mathbf{g}_8^{(4)} \\ \mathbf{g}_9^{(4)} \\ \mathbf{g}_{10}^{(4)} \\ \mathbf{g}_{11}^{(4)} \\ \mathbf{g}_{12}^{(4)} \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \ldots\ldots \quad (5.29)$$

All the old zero elements in the array $\mathbf{G}^{(2)}$ have been replaced with the new non-zero components of $\mathbf{g}^{(2)}$, $\mathbf{g}^{(6)}$, $\mathbf{g}^{(10)}$, etc. There is no risk of overlapping, since the vector $\mathbf{G}^{(2)}$ will remain unchanged throughout all the successive iterations.

We start the second iteration, using $\mathbf{G}^{(2)}$ to update $\mathbf{G}^{(3)}$, $\mathbf{G}^{(4)}$,..., $\mathbf{G}^{(q-1)}$. At the end of iteration 2, vector $\mathbf{G}^{(3)}$ will be filled by the new non-zero components of $\mathbf{g}^{(4)}$, $\mathbf{g}^{(12)}$, etc. Again, there is no risk of overlapping, since the vector $\mathbf{G}^{(3)}$ will remain unchanged throughout all the successive iterations.

During the third iteration, the values stored in $\mathbf{G}^{(3)}$ are used to update all the following $\mathbf{G}^{(j)}$ for $j = 4, 5, \ldots, q-1$. The whole process is repeated until the $(q-1)^{th}$ iteration has been completed.

With each iteration of Stage 3, while updating the vectors $\mathbf{G}^{(j)}$, we will also have updated the component values of $\mathbf{u}$, by updating the corresponding subvectors $\mathbf{u}_{i-(i+k)}$ (see figure (5.17)). After the last iteration, therefore, we will have built the final solution $\mathbf{u} = \mathbf{u}_{1-m}$, as a vector of n components.

Finally, let us consider each vector $\mathbf{G}^{(j)}$ as the j^{th} column of a single matrix G of dimension $nx(q-1)$

$$G = (\,\mathbf{G}^{(1)} \quad \mathbf{G}^{(2)} \quad \mathbf{G}^{(3)} \quad \ldots \quad \mathbf{G}^{(q-1)}\,) \tag{5.30}$$

In this way we have built a structure that avoids the above mentioned overlapping of components. The matrix G also simulates the Fan-in tree graph, enabling us to perform the updating calculations in parallel. The form of matrix G explains the reason why the iteration index j ranges from 1 to $q-1$, instead of ranging from 1 to $m-1$: by introducing G, in fact, we have in some way replaced vectors $\mathbf{g}^{(j)}$ where $j = 1, 2, \ldots, m-1$ with vectors $\mathbf{G}^{(j)}$ where $j = 1, 2, \ldots, q-1$.

Since vectors $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$ are used respectively to define and update vectors $\mathbf{g}^{(j)}$, we have chosen to store them in a structure similar to the matrix G. Therefore, all the vectors $\mathbf{x}^{(j)}$ are stored in a matrix of dimension $nx(q-1)$, as shown in the following figure

$$X = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \\ 1 & 0 & 0 & \\ 0 & 0 & 1 & \\ 0 & 0 & 1 & \cdots \\ 1 & 0 & 0 & \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 1 & 0 & \cdots \\ 1 & 0 & 0 & \\ 1 & 0 & 0 & \\ 0 & 0 & 0 & \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{5.31}$$

Since the values of vectors $\mathbf{x}^{(j)}$ are not needed any longer after the end of Stage 2, the same matrix X is used to overwrite the values of vectors $\mathbf{y}^{(j)}$.

During Stage 3, X is then defined as follows:

$$X = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ a_3 & 0 & 0 & \\ c_2 & 0 & 0 & \\ 0 & a_5 & 0 & \\ 0 & c_4 & 0 & \cdots \\ a_7 & 0 & 0 & \\ c_6 & 0 & 0 & \\ 0 & 0 & a_9 & \\ 0 & 0 & c_8 & \cdots \\ a_{11} & 0 & 0 & \\ c_{10} & 0 & 0 & \\ 0 & a_{13} & 0 & \\ 0 & c_{12} & 0 & \cdots \\ a_{15} & 0 & 0 & \\ c_{14} & 0 & 0 & \\ 0 & 0 & 0 & \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{5.32}$$

Now that we have justified the use of the two particular structures G and X, we carry on with the description of the Recursive Decoupling routine.

After reading the input data (i.e. the exponent q defining the system size and the number of processors we want to use), all the arrays and the matrices are initialised either to zero or to their current values. Note that, in the Fortran program, G and X are respectively memorized in the two equivalent matrices g and x.

The Preliminary Stage follows this initialisation section, performing the requested partitioning (5.3) of the coefficient matrix. In practice, during the Pre-stage we explicity calculate the 2x2 submatrices J_i^{-1}.

After this, Stage 1, Stage 2, Stage 3 implement all the operations and calculations described in the paragraph 5.3.

Note that the matrix x, initialized to zero, is assigned to store the values of vectors $\mathbf{x}^{(j)}$ during Stage 2, and then to store the values of vectors $\mathbf{y}^{(j)}$ during Stage 3. There is no actual initialisation of the matrix x to the values of the $\mathbf{x}^{(j)}$. As a consequence of the definition of the same vectors $\mathbf{x}^{(j)}$, in fact, the operations described in formula (5.22) can be carried out by using directly the values of matrices J_i^{-1}, as shown below:

$$\begin{pmatrix} g_{2i-1}^{(j)} \\ g_{2i}^{(j)} \end{pmatrix} = J_i^{-1} \begin{pmatrix} x_{2i-1}^{(j)} \\ x_{2i}^{(j)} \end{pmatrix} = J_i^{-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 2^{nd}\text{column of } J_i^{-1}$$

$$\begin{pmatrix} g_{2i+1}^{(j)} \\ g_{2i+2}^{(j)} \end{pmatrix} = J_i^{-1} \begin{pmatrix} x_{2i+1}^{(j)} \\ x_{2i+2}^{(j)} \end{pmatrix} = J_i^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1^{st}\text{column of } J_i^{-1}$$

In order to perform the operations (4.2) and initialise the vectors $\mathbf{g}^{(j)}$ during Stage 2, therefore, we simply work on the columns of matrices J_i^{-1}. At the end of this second stage, we directly assign to the matrix x the values of vectors $\mathbf{y}^{(j)}$.

At this point, having taken account of all the above considerations and observations and having chosen the described structures of arrays and matrices, the parallelism in the implemented Fortran program is easily exploited.

Since the Recursive Decoupling routine is intended to be run on the Balance 8000 multiprocessor, we have made use of the programming tools available on the Sequent systems [12]. More specifically, we have used both the *Fortran Parallel Programming Directives* and the *Parallel Programming Library.*

After the initialisation phase and before starting the Pre-stage, we use the **m_set_procs** microtasking routine to declare the number of processors assigned to our task (up to 9 processors). The Pre-stage, Stage 1 and Stage 2 can be parallelized by simply using the **Doacross** parallel directive, since they consist of an independent loop, namely a loop in which no iteration depends the calculations in any other iteration. Since Stage 3 shows a higher complexity, we need to use the routines from the Parallel Programming Library, organizing all the operations into a subroutine (*stage* 3) that will be performed in parallel.

We call the **m_fork** microtasking routine to fork the set number of child processes and assign them to the subroutine stage 3. Depending on which iteration of the third stage we are considering, each forked child process 'calls' the subroutine stage 3 and performs the updating of array **u** and matrix g, working on different subsets of components of the same **u** and g. The size of these subsets varies according to the current iteration number, namely

$$size = 2^{(k+1)} \text{ during the } k^{th} \text{ iteration}$$

Different child processes work on different subsets of components, so that the operations performed by one process are totally independent from the calculations done by another process, though all of them are updating the same array **u** and matrix g.

For example, during iteration $k = 1$ the size of each components subset is equal to 4 and therefore:

- the first child process updates the first 4 elements of **u** and the first 4 rows of g;
- the second child process updates components from 5 to 8 of **u** and rows from 5 to 8 of g;
- and so on.

The index of the current component (or row) where each single child process has to start updating is given by the subroutine *partition*. According to the values of two logical variables *go* and *var*, the subroutine partition assigns the appropriate value of the updating starting index; this value is stored in the integer variable *begin*.

The variable *go* states whether there are more components to be updated (*go* = *true*) or if the work concerning the current iteration has been completed (*go* = *false*).

The logical variable *var* is a flag control for the particular case: *begin* = *1*. The use of *var* is necessary to re-initialise to 1 the value of *begin* at the beginning of each new iteration. Starting one iteration, we have *var* = *true*. After the first child process has called the subroutine partition and initialised *begin* = *1*, the logical value of *var* becomes *false* and it remains false for each successive calling processes, until the iteration has been completed.

The subroutine *partition* is written inside a 'locked region', to avoid the possibility of two child processes calling this subroutine at the same time and therefore accessing the same shared value of *begin*. To create this protected region in the Fortran code, we have used the **m_lock** and the **m_unlock** microtasking routines.

Since the number of the current k^{th} iteration is essential to decide the value of variables *begin* and *size*, the iteration number k is given as an argument of the subroutine stage 3.

We can schematize the parallel operations performed in Stage 3 as follows:

- a set number of child processes is created and the current iteration number k is passed to subroutine stage 3;
- each child process 'goes' to the subroutine partition and is given its own updating starting index;
- each process performs all the operations on the appropriate subset of components of vector **u** and of matrix g;
- when its task is finished, each child process 'goes' again to subroutine *partition* to find out if there is more work to do;
- when the k^{th} iteration has been completed, we call the **m_sync** microtasking routine, to synchronize all the processes and to assure they have all finished the tasks related to iteration k;
- finally, when all the $q-1$ iterations have been terminated, we 'kill' all the forked processes by using the **m_kill_procs** routine from the Parallel Programming Library.

At this point, we write all the output data, i.e. the dimension problem parameters, the obtained solution **u**, the error between the exact solution and **u**, the computing time (given in seconds).

The routine used to time our program is the *_clock_time* routine, written in C language.

The two subroutines *dmatvet* and *matvet*, in our Fortran program, are used respectively during Stage 1 and Stage 3 to perform matrix/vector multiplications.

Before concluding this description of the Recursive Decoupling routine, we need to make one more observation.

The last iteration of Stage 3 is performed serially, by using the subroutine *stage* 33. During this final iteration, in fact, the following values

$$k = q - 1 \qquad\qquad size = 2^q = n$$

are assigned. This means that, even if performing this iteration in parallel, all the work is done by only one process, since the current subset of components of $\mathbf{u}$ and g coincides with the whole vector $\mathbf{u}$ and the whole matrix g.

Since the implemented Fortran routine is synchron**ous** and no matter what the dimension of the tridiagonal linear system, the calculations involved in the last iteration are carried out by only one processor. Therefore, we can proceed in two ways:

i) perform $q-2$ iterations in parallel and then the last one serially, using two different subroutines (stage 3 and stage 33), to implement the parallel code and the serial code;

ii) perform all the $q-1$ iterations in parallel, using the subroutine stage 3 only, but leaving all the child processes except one in a spinning state during the last iteration.

The above mentioned two ways have been implemented in two different versions (version *Parallel.f* and version *Paralleli.f* respectively) of the Recursive Decoupling routine. By testing them on different coefficient matrices, it has been proved that choice (i) is slightly less efficient than choice (ii) in terms of elapsed time (see tables of results in the next paragraph).

There is a third possible way of performing the iterations of Stage 3, which is based on the following observation: no matter what the dimension of the system, the second last iteration is always performed by 2 child processes and the third last iteration by 4 child processes. Therefore we can:

- use a maximum number of processors to perform iterations from 1 to $q-4$;
- use 4 processors during the $(q-3)^{th}$ iteration;
- use 2 processors during the $(q-2)^{th}$ iteration;
- use 1 processor during the $(q-1)^{th}$ iteration (the last one).

The above scheme, though, involves several calls to the **m_set_procs**, **m_fork** and **m_kill_procs** routines. This is relatively expensive in terms of the overall computational cost. Thus, the version of the Recursive Decoupling routine implementing this third choice (version *Parallelo.f*), therefore, is more time consuming than the previous two versions.

These results are shown in more detail in the following paragraph 5.8.

5.8. Numerical Experiments and Remarks

In this paragraph numerical results are reported, concerning the solution of tridiagonal linear systems by means of the Recursive Decoupling routine, on the Balance 8000 parallel machine.

Similar to the Wang routine (see paragraph 4.4), the following tables group together the execution timing (both for the sequential and the parallel versions of the algorithm), the experimental speed-up (to be compared with the expected speed-up) and the efficiency parameters.

The maximum error E_{max}, average error E_{av} and maximum relative error E_r are also presented, in order to study the degree of accuracy obtained by the Recursive Decoupling method. These error measurements have been calculated according to formulae (4.4).

All the results shown are related to the two test tridiagonal systems presented in paragraph 4.4 (see figure 4.5 and figures 4.6 & 4.7).

First of all, the accuracy results are shown in the following table 5.33. These results are general to all the versions of the Recursive Decoupling Fortran program, including the sequential version. Unlike the Wang routine, the accuracy obtained when using the Recursive Decoupling routine to solve the first example does not depend on the number of processors used. In this case the random properties observed in the accuracy results of the Wang program does not occur.

TABLE 5.33

	Maximum Error E_{max}	Average Error E_{av}	Maximum Relative Error E_r
n=64 q=6	.0000005364	.0000001230	.0000005448
n=128 q=7	.0000021458	.0000011813	.0000021626
n=256 q=8	.0000027418	.00000122282	.0000027525
n=512 q=9	.0000067949	.0000036428	.0000068082
n=1024 q=10	.0000085235	.0000038184	.0000085318

Accuracy results obtained when using any version of the Recursive Decoupling routine to solve the first example system (4.5).

As in the case of the Wang method, 100% accuracy is reached by the Recursive Decoupling routine in the solution of the second test system.

The following tables show the execution times obtained when running the R. D. routine on the two test examples. The speed-up and efficiency parameters are also calculated and reported in the same tables. Both the elapsed times and the reclaimed times are given in different sets of tables for each one of the three versions of the Recursive Decoupling routine (for the definition of "reclaimed time" see paragraph 4.4).

The notation adopted is as for the Wang algorithmic results. For what concerns accuracy measurements. Speed-up and Efficiency are computed as described in chapter 3.
A slight variation occurs in the results which is probably due to overheads arising from manipulation and allocation of different stack sizes.

TABLE 5.34

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	1.636000	.8325	2.00	.2081
n=128 q=7 p=4	5.4500	4.396000	1.2398	2.00	.3099
n=256 q=8 p=4	23.6990	15.278500	1.5511	2.00	.3878
n=512 q=9 p=4	112.2330	58.438510	1.9205	2.00	.4801
n=1024 q=10 p=4	493.0700	230.670000	1.9035	2.00	.4759

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	2.229000	.6110	2.6667	.0764
n=128 q=7 p=8	5.4500	4.932500	1.1049	2.6667	.1381
n=256 q=8 p=8	23.6990	15.612500	1.5180	2.6667	.1897
n=512 q=9 p=8	112.2330	57.832990	1.9406	2.6667	.2426

Elapsed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallel.f, to solve the first example system (4.5).

TABLE 5.35

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	.91250	1.4926	2.00	.3732
n=128 q=7 p=4	5.4500	3.60000	1.5139	2.00	.3785
n=256 q=8 p=4	23.6990	14.23700	1.6646	2.00	.4162
n=512 q=9 p=4	112.2330	57.39000	1.9556	2.00	.4889
n=1024 q=10 p=4	439.0700	228.78000	1.9192	2.00	.4798

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	.89650010	1.5192	2.6667	.1899
n=128 q=7 p=8	5.4500	3.49500000	1.5594	2.6667	.1949
n=256 q=8 p=8	23.6990	13.66250000	1.7346	2.6667	.2168
n=512 q=9 p=8	112.2330	55.30500000	2.0293	2.6667	.2537

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallel.f, to solve the first example system (4.5).

TABLE 5.36

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	1.6320	.8346	2.00	.2086
n=128 q=7 p=4	5.4500	4.3940	1.2403	2.00	.3101
n=256 q=8 p=4	23.6990	15.2640	1.5526	2.00	.3882
n=512 q=9 p=4	112.2330	58.4970	1.9186	2.00	.4797
n=1024 q=10 p=4	439.0700	230.4500	1.9053	2.00	.4763

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	2.2230000	.6127	2.6667	.0766
n=128 q=7 p=8	5.4500	4.9309990	1.1053	2.6667	.1382
n=256 q=8 p=8	23.6990	15.5980000	1.5194	2.6667	.1899
n=512 q=9 p=8	112.2330	57.9010000	1.9384	2.6667	.2423

Elapsed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallel.f, to solve the second example system (4.6 & 4.7).

TABLE 5.37

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	.908000000	1.5000	2.00	.3750
n=128 q=7 p=4	5.4500	3.594000000	1.5164	2.00	.3791
n=256 q=8 p=4	23.6990	14.215000010	1.6672	2.00	.4168
n=512 q=9 p=4	112.2330	57.327000000	1.9578	2.00	.4894
n=1024 q=10 p=4	439.0700	228.430000000	1.9221	2.00	.4805

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	.88000020	1.5477	2.6667	.1935
n=128 q=7 p=8	5.4500	3.47900100	1.5665	2.6667	.1958
n=256 q=8 p=8	23.6990	13.77000000	1.7211	2.6667	.2151
n=512 q=9 p=8	112.2330	55.39700000	2.0260	2.6667	.2532

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallel.f, to solve the second example system (4.6 & 4.7).

TABLE 5.42

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	1.9460000	.6999	2.00	.1750
n=128 q=7 p=4	5.4500	4.7220010	1.1542	2.00	.2885
n=256 q=8 p=4	23.6990	15.7430000	1.5054	2.00	.3763
n=512 q=9 p=4	112.2330	59.927500	1.8728	2.00	.4682
n=1024 q=10 p=4	493.0700	234.7800000	1.8701	2.00	.46750

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	2.539000	.5364	2.6667	.0671
n=128 q=7 p=8	5.4500	5.262000	1.0357	2.6667	.1295
n=256 q=8 p=8	23.6990	16.006000	1.4806	2.6667	.1851
n=512 q=9 p=8	112.2330	59.172010	1.8967	2.6667	.2371

Elapsed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallelo.f, to solve the first example system (4.5).

TABLE 5.43

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	.91000010	1.4967	2.00	.3742
n=128 q=7 p=4	5.4500	3.60500000	1.5118	2.00	.3779
n=256 q=8 p=4	23.6990	14.37000000	1.6492	2.00	.4123
n=512 q=9 p=4	112.2330	57.49000000	1.9522	2.00	.4881
n=1024 q=10 p=4	439.0700	227.89000000	1.9267	2.00	.4817

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	.89000030	1.5303	2.6667	.1913
n=128 q=7 p=8	5.4500	3.49800000	1.5580	2.6667	.1948
n=256 q=8 p=8	23.6990	13.88400000	1.7069	2.6667	.2134
n=512 q=9 p=8	112.2330	55.42699000	2.0249	2.6667	.2531

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallelo.f, to solve the first example system (4.5).

TABLE 5.44

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	1.9400	.7021	2.00	.1755
n=128 q=7 p=4	5.4500	4.7280	1.1527	2.00	.2882
n=256 q=8 p=4	23.6990	15.6560	1.5137	2.00	.3784
n=512 q=9 p=4	112.2330	59.8620	1.8749	2.00	.4687
n=1024 q=10 p=4	439.0700	234.1900	1.8748	2.00	.4687

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	2.5300	.5383	2.6667	.0673
n=128 q=7 p=8	5.4500	5.2560	1.03691	2.6667	.1296
n=256 q=8 p=8	23.6990	15.9860	1.4825	2.6667	.1853
n=512 q=9 p=8	112.2330	59.1970	1.8959	2.6667	.2370

Elapsed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallelo.f, to solve the second example system (4.6 & 4.7).

TABLE 5.45

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=4	1.3620	.918000100	1.4837	2.00	.37090
n=128 q=7 p=4	5.4500	3.610000000	1.5097	2.00	.3774
n=256 q=8 p=4	23.6990	14.233000010	1.6651	2.00	.4163
n=512 q=9 p=4	112.2330	57.390000000	1.9556	2.00	.4889
n=1024 q=10 p=4	439.0700	228.360000000	1.92270	2.00	.4807

	T_s	T_p	Obtained $S_p(n)$	Expected $S_p(n)$	Efficiency $E_p(n)$
n=64 q=6 p=8	1.3620	.8980	1.5167	2.6667	.1896
n=128 q=7 p=8	5.4500	3.4990	1.5576	2.6667	.1947
n=256 q=8 p=8	23.6990	13.7700	1.7211	2.6667	.2151
n=512 q=9 p=8	112.2330	55.4100	2.0255	2.6667	.2532

Reclaimed times (in seconds), speed-up and efficiency obtained when using the Recursive Decoupling routine, version Parallelo.f, to solve the second example system (4.6 & 4.7).

6. Conclusions and Further Work

6.1. Conclusions and Suggestions for Further Work

In this thesis we have presented a new method for solving tridiagonal systems of linear equations and we have used the Wang method as a term of reference.

Both algorithms belong to the "partitioning" class methods, but they have very different behaviour.

The Wang routine, though extremely fast in term of execution times, shows negative characteristics, such as a rapid decrease in accuracy results as the dimension of the problem increases.

This is due to the amount of data transferred between the processors of a parallel computer (as in the Balance 8000). The partitioning process in the Wang method, in fact, is such that the elimination process inside one group of k rows ($k = n/p$) requires more than just one processor. That is to say, the p subsytems created by the partitioning process are not independent from each other (we recall that n indicates the dimension of the problem, while p is the number of processors used) [20].

The Recursive Decoupling routine cannot compete with the Wang routine in terms of execution times: the new method we presented is much slower than that of Wang.

On the other hand, the Recursive Decoupling method shows better behaviour in terms of accuracy, the decrease in accuracy being much less as the dimension increases.

The memory allocation requirement for the Recursive Decoupling routine is not favourable, compared to the same requirement in the Wang routine.

This is due to the problem discussed in paragraph 5.7, concerning the implementation of the tree structure of figure 5.17.

We suggest that an implementation of the Recursive Decoupling method in other programming languages, such as Pascal or C, would probably result in an improvement in the memory allocation requirements. The same improvement would probably be possible by using the new version of Fortran compiler ("Fortran 90"), which allows the programmer to allocate memory dynamically and permits direct operations on data structures; such as direct matrix multiplication and scalar product of vectors.

In terms of speed-up and efficiency, the two methods under comparison can be considered equivalent.

We would like to underline that the main characteristic of the Recursive Decoupling method is given by its partitioning process, which leads to independent subsystems (that is to say, it leads to independent sets of data to be assigned to each single processor).

The relative simplicity of the formulae involved should also be considered. The partitioning of the original linear system into 2×2 independent subsystems allows the immediate and explicit expression of the inverse matrix.

On the other hand, the same partitioning process results in high execution times; the reason being that the calculation performed by each processor is too small, compared to the overhead involved in the creation of multiple processes.

As a consequence of all these considerations, a suggestion for further research could consist of increasing the size of the subsystems into which the original

system is partitioned. By recursively partitioning the given tridiagonal system into 4×4 or even higher dimension independent subsystems, the calculation performed by each processor becomes more substantial; that is to say, the overhead due to multiple process creation becomes less relevant.

This different way of performing the partitioning process (in the Recursive Decoupling algorithm) will certainly lead to a loss in the simplicity of the formulae involved.

The balance between positive and negative aspects of this possible partitioning process is currently under investigation; this development of the Recursive Decoupling method can be considered as the core for further research.

Other possible developments of the method presented are represented by the adaptation of the routine to special cases, such as the solution of tridiagonal linear systems with constant sub-diagonal, constant main diagonal and constant upper diagonal matrix elements (i.e. Toeplitz systems).

To conclude, we mention the fact that, often, many independent tridiagonal linear systems need to be solved. We can then take advantage of the independence of the systems, as well as the independent characteristics of the partitioning process, and apply the Recursive Decoupling algorithm to a single large tridiagonal system.

For example, if we need to solve two systems of dimension 3×3 and 4×4 respectively, we can consider the solution of a single system of dimension 7×7, as follows:

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & 0 & & & \\ & & 0 & B_1 & C_1 & & \\ & & & A_2 & B_2 & C_2 & \\ & & & & A_3 & B_3 & C_3 \\ & & & & & A_4 & B_4 \end{pmatrix}$$

When considering this "linking" of parallel systems, the balance between the performance gain (which is directly proportional to the increase of the problem dimension) and the memory allocation requirement must be investigated.

References

1. Bekakos, M. P., *"Design and Analysis of Parallel Algorithms"*, Thesis for the Degree of Master of Science in Theory and Application of Computation, Supervisor Prof. D. J. Evans, Loughborough University of Technology, Loughborough, Leics., U.K., September 1981.

2. Brawer, S., *"Introduction to Parallel Programming"*, Encore Computer Corporation, Marlborough, Massachussetts, Academic Press Inc., 1989.

3. Cowell, W. R., *"User's Guide to Toolpack/1. Tools for Data Dependency Analysis and Program Transformation"*, Preprint ANL-88-17, Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago, Chicago, U.S.A., September 1988.

4. Cowell, W. R., Thompson, C. P., *"Tools to Aid in Discovering Parallelism and Localising Arithmetic in Fortran Programs"*, Preprint MCS-P6-1088, Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago, Chicago, U.S.A., October 1988.

5. Evans, D. J., "A Recursive Decoupling Method for Solving Tridiagonal Linear Systems", *International Journal Comp. Maths.*, **33**, 1990, pp.95-102.

6. Evans, D. J., *"Parallel Algorithm Design"*, Computer Studies report 463, Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leics., U.K., November 1988.

7. Evans, D. J., "Parallel Numerical Algorithms for Linear Systems", in *Parallel Processing Systems*, ed. D. J. Evans, Cambridge University Press, 1982, pp.357-383.

8. Evans, D. J., *"A Comparison of Sequential and Parallel Elimination Methods for Tridiagonal Matrices"*, Computer Studies report 436, Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leics., U.K., May 1988.

9. Evans, D. J., *"Multitasking Strategies in Parallel Computing"*, Computer Studies report 557, Parallel Algorithms Research Centre, Loughborough University of Technology, Loughborough, Leics., U.K., October 1990.

10. Evans, D. J., Forrington, C. V. D., "Note on the Solution of Certain Tridiagonal Systems of Linear Equations", *Computer Journal*, **5**, 1963, pp.327-328.

11. Evans, D. J., Hatzpopoulos, "A parallel Linear System Solver", *International Journal Comp. Maths.*, **7**, 1979, pp.227-238.

12. Evans, D. J., Wheat, M., *"Parallel Processing on the Balance 8000 Computer"*, P.A.R.C. Internal Memoranda, Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leics., U.K., 1989.

13. Flynn, M. J., "Very High Speed Computing Systems", *Proc. I.E.E.E.*, **14**, 1966, in [23], p.17.

14. Gregory, R. T., Karney, D. L., *"A Collection of Matrices for Testing Computational Algorithms"*, Wiley-Interscience, a division of John Wiley & Sons, 1969, pp.40-52.

15. Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra", *S.I.A.M. Review*, **20**, 4, October 1978, in [23], p. 35.

16. Karp, A. H., "Programming for Parallelism", *Computer*, 23, May 1987, pp.43-57.

17. Kogge, P. M., " Parallel Solution of Recurrence Problems", *IBM J. Res. Develop.*, **18**, March 1974, pp.138-148, in [23], p.36.

18. Kowalik, J. S., Lord, R. E., Kumar, S. P., "Design and Performance of Algorithms for MIMD Parallel Computers. Tridiagonal Linear Equations", *High-Speed Computation*, ed. J. S. Kowalik, NATO A.S.I. Series, Springer-Verlag Berlin/Heidelberg/New York/Tokio, 1984, pp.267-275.

19. Kuck, D. J., Muraoka, Y., "Bounds on the Parallel Evaluation of Arithmetic Expressions using Associativity and Commutativity", *Acta Informatica*, **3**, 3, 1974, pp.203-216, in [23], p.31.

20. Michielse, P. H., Van Der Vorst, H. A., *"Data Transport in Wang's Partition Method"*, Report 86-32, Dept. of Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands, 1986.

21. Reuter, R., Zecca, V., "A Parallel Method for Solving Tridiagonal Systems of Linear Equations on the IBM 3090 Vector Multiprocessor", *High Performance Computing*, J. L. Delhaye & E. Gelenbe editors, Elsevier Science Publishers B. V. (North Holland), 1989, pp.33-43.

22. Ruggiero, V., Duri', F., "Analisi di Algoritmi per la Risoluzione di Sistemi Tridiagonali su un Calcolatore Vettoriale", *Atti dell'Accademia delle Scienze dell'Istituto di Bologna*, Bologna, Italia, 1989.

23. Schendel, U., *"Introduction to Numerical Methods for Parallel Computers"*, Ellis Horwood Ltd., 1984.

24. Sequent Computer Systems, *"Guide to Parallel Programming on Sequent Computer Systems"*, 1987.

25. Sequent Computer Systems, *"Dynix Fortran Compiler User's Manual"*, Man-0510-00, March, 21, 1986.

26. Sequent Computer Systems, *"Balance 8000 System Technical Summary"*, Man-0110-00, December 12, 1985.

27. Sherman, J., Morrison, W. J., "Adjustment of an Inverse Matrix Corresponding to Changes in a Given Column or a Given Row of the Original Matrix", *Ann. Math. Stal.*, **20**, 1949, pp.621.

28. Stone, H. S., "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", *J.A.C.M.*, **20**, 1, January 1973, pp.27-38.

29. Stone, H. S., "Parallel Tridiagonal Equations Solvers", *A.C.M. Transactions on Mathematical Software*, **1**, 4, December 1975, pp.289-307.

30. Stone, H. S., "Problems of Parallel Computation", *Complexity of Sequential and Parallel Numerical Algorithms*, ed. J. F. Traub, Academic Press, 1973, in [23], p.13.

31. Wang, H. H., "A Parallel Method for Tridiagonal Equations", *A.C.M. Transactions on Mathematical Software*, **57**, 2, June 1981, pp.170-183.

Appendix. Further Numerical Experiments

In the following pages additional results are obtained by testing the Recursive Decoupling routine on further tridiagonal linear systems $A\mathbf{x} = \mathbf{d}$.

Since the exact solution $\mathbf{x}$ is, in general, not known, the solution accuracy is studied by means of "residual" results, that is to say once the computed solution $\mathbf{u}$ has been obtained, we form the residual vector

$$\mathbf{r} = \mathbf{d} - A\mathbf{u}. \tag{A.1}$$

The following tables show respectively the quantities:

$$\begin{aligned} u_{max} &= \| \mathbf{u} \|_\infty \\ Res_{max} &= \| \mathbf{r} \|_\infty \\ Res_{av} &= \frac{\sum_{i=1}^{n} |r_i|}{n} \end{aligned} \tag{A.2}$$

where n is the problem dimension ($n = 4, 8, 16, ..., 1024$).

Results in brackets are obtained by testing the Wang routine on the same tridiagonal linear systems considered in this appendix. Note that in the case of the Wang routine, results depend on the number p of processors used (also shown in brackets) unlike the Recursive Decoupling routine results, which are independent of p.

Test system 3

The third tridiagonal linear system considered is

$$\begin{pmatrix} 2.05 & -1 & & & & 0 \\ -1 & 2.05 & -1 & & & \\ & -1 & 2.05 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2.05 & -1 \\ 0 & & & & -1 & 2.05 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n-1 \\ n \end{pmatrix} \qquad (A.\ 3)$$

TABLE EXAMPLE 3

	Calculated Sol. u_{max}	Max. Residual Res_{max}	Average Residual Res_{av}
n=4, q=2	7.096779	.0000019073	.0000007153
(p=2)	(7.096777)	(.0000009537)	(.0000002384)
n=8, q=3	32.98121	.0000095367	.0000038147
(p=2)	(32.98119)	(.0000038147)	(.0000014305)
n=16, q=4	131.4839	.0000457764	.0000162125
(p=4)	(131.4838)	(.0000152588)	(.0000076294)
n=32, q=5	391.4189	.0000915527	.0000259876
(p=8)	(391.4184)	(.0000915527)	(.0000230074)
n=64, q=6	970.6665	.0002441406	.0000594854
(p=8)	(970.6656)	(.0002441406)	(.0000512004)
n=128, q=7	2189.229	.0009765625	.0001681149
(p=8)	(2189.226)	(.0004882812)	(.0001171231)
n=256, q=8	4687.417	.0019531250	.0003601462
(p=8)	(4687.410)	(.0019531250)	(.0003023446)
n=512, q=9	9745.392	.0039062500	.0007368997
(p=8)	(9745.369)	(.0039062500)	(.0004862323)
n=1024, q=10	19923.24	.0097656250	.0015076140
(p=8)	(19923.19)	(.0068359370)	(.0007717274)

Test system 4

The fourth tridiagonal linear system considered is

$$\begin{pmatrix} 2.05 & 1 & & & & 0 \\ -1 & 2.05 & 1 & & & \\ & -1 & 2.05 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2.05 & 1 \\ 0 & & & & -1 & 2.05 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n-1 \\ n \end{pmatrix} \qquad (A.\ 4)$$

TABLE EXAMPLE 4

	Calculated Sol. u_{max}	Max. Residual Res_{max}	Average Residual Res_{av}
n=4, q=2	2.28882	.0000002384	.0000000596
(p=2)	(2.28882)	(.0000004768)	(.0000001788)
n=8, q=3	5.020085	.0000004768	.0000001564
(p=2)	(5.020085)	(.0000009537)	(.0000002086)
n=16, q=4	10.5104	.0000019073	.0000004582
(p=4)	(10.5104)	(.0000019073)	(.0000006407)
n=32, q=5	21.49187	.0000038147	.0000012945
(p=8)	(21.49187)	(.0000057220)	(.0000010058)
n=64, q=6	43.4548	.0000114441	.0000029271
(p=8)	(43.4548)	(.0000076294)	(.0000015572)
n=128, q=7	87.38066	.0000228882	.0000057551
(p=8)	(87.38067)	(.0000305176)	(.0000037495)
n=256, q=8	175.2324	.0000610352	.0000100897
(p=8)	(175.2324)	(.0000610352)	(.0000068760)
n=512, q=9	350.9358	.0001220703	.0000187539
(p=8)	(350.9358)	(.0000915527)	(.0000120304)
n=1024, q=10	702.3428	.0002441406	.0000362289
(p=8)	(702.3428)	(.0001831055)	(.0000258710)

Test system 5

The fifth tridiagonal linear system considered is

$$\begin{pmatrix} 2.02 & -2 & & & & 0 \\ -2 & 2.02 & -2 & & & \\ & -2 & 2.02 & -2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -2 & 2.02 & -2 \\ 0 & & & & -2 & 2.02 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n-1 \\ n \end{pmatrix} \qquad (A.\ 5)$$

TABLE EXAMPLE 5

	Calculated Sol. u_{max}	Max. Residual Res_{max}	Average Residual Res_{av}
n=4, q=2	3.511341	.0000009537	.0000004172
(p=2)	(3.511341)	(.0000009537)	(.0000003576)
n=8, q=3	79.42160	.00000324249	.00000214875
(p=2)	(79.42368)	(.0002746582)	(.0000411421)
n=16, q=4	15.58781	.0000114441	.0000049174
(p=4)	(15.58777)	(.0000801086)	(.0000123680)
n=32, q=5	97.00327	.0000915527	.0000297353
(p=8)	(97.00712)	(.0004806519)	(.0000590049)
n=64, q=6	64.05173	.0000762939	.0000200570
(p=8)	(64.05173)	(.0004043579)	(.0000373274)
n=128, q=7	157.22079	.0001373291	.0000356380
(p=8)	(157.22080)	(.0011291500)	(.0000651926)
n=256, q=8	354.9340	.0003356934	.0000830218
(p=8)	(354.9276)	(.0027465820)	(.0000877231)
n=512, q=9	1721.093	.0021972660	.0005718022
(p=8)	(1720.767)	(.0129394500)	(.0005405098)
n=1024, q=10	1319.145	.0068359370	.0006879344
(p=8)	(1319.226)	(.0086669940)	(.0002056956)

Test system 6

The sixth tridiagonal linear system considered is

$$\begin{pmatrix} 2 & -2 & & & & 0 \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ 0 & & & & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n-1 \\ n \end{pmatrix} \qquad (A.\ 6)$$

TABLE EXAMPLE 6

	Calculated Sol. u_{max}	Max. Residual Res_{max}	Average Residual Res_{av}
n=4, q=2	18.0	.000009537	.0000007153
(p=2)	(18.0)	(.000000000)	(.0000000000)
n=8, q=3	115.9999	.000076294	.0000014305
(p=2)	(116.0000)	(.000038147)	(.0000004768)
n=16, q=4	807.9993	.0001220703	.0000343323
(p=4)	(808.0008)	(.0001220703)	(.0000238419)
n=32, q=5	5968.004	.0014648440	.0002326966
(p=8)	(5968.032)	(.000976525)	(.0002727509)
n=64, q=6	45727.85	.0039062500	.0010986330
(p=8)	(45727.98)	(.0078125000)	(.0015106200)
n=128, q=7	357695.6	.0937500000	.0106124900
(p=8)	(357702.8)	(.0937500000)	(.0222244300)
n=256, q=8	2828965.0	.5000000000	.0536651600
(p=8)	(2828835.0)	(.7500000000)	(.1765747000)
n=512, q=9	22500730.0	5.0000000000	1.0875240000
(p=8)	(22496650.0)	(8.0000000000)	(1.5995480000)
n=1024, q=10			

From the given examples we can see that for matrices with reasonable P condition number (e.g. test system 4) the accuracy given by the Recursive Decoupling algorithm is $10^{-4} \div 10^{-6}$, even for 9 levels of recursions.

However for the ill conditioned examples $3, 5, 6$ it is clear that the rounding errors do increase dramatically with increasing recursion levels; this affects the obtained accuracy, suggesting that in these cases the maximum number of recursion levels is $4 \div 5$ in order to achieve solution accuracy of 10^{-4}.

In these cases it would be advisable to stop the recursion at level 4 (or 5) and proceed to solve smaller tridiagonal subsystems by the Gaussian elimination algorithm (version for tridiagonal systems).

In order to achieve solution accuracy in the case of ill conditioned or quasi-singular systems, it could also be necessary to perform the calculations in double/multiple precision arithmetic. This would account for more costs in terms of memory requirements and computing time.

The numerical results contained in this appendix confirm all the conclusions in the thesis concerning the Wang algorithm and the Recursive Decoupling algorithm.

Appendix. Programs Listings

Program Decoupling. Version Parallel.f

This program solves tridiagonal linear systems $A\mathbf{u} = \mathbf{d}$ using a recursive decoupling technique. The system considered is of the form

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & & & & \\ & 0 & & & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \\ \vdots \\ d_n \end{pmatrix}$$

This solution routine is formulated into a preliminary stage and then into three different sections.

Preliminary Stage or Pre-stage

The coefficient matrix A is rearranged into the following form

$$\begin{pmatrix} e_1 & c_1 & & & & & \\ a_2 & e_2 & & & & 0 & \\ & & e_3 & c_3 & & & \\ & & a_4 & e_4 & & & \\ & & & & \ddots & & \\ & 0 & & & & e_{n-1} & c_{n-1} \\ & & & & & a_n & e_n \end{pmatrix} + \sum_{j=1}^{m-1} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}^{(j)} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix}^{(j)T}$$

where

$$e_1 = b_1$$

$$e_{2j-1} = b_{2j-1} - c_{2j-2} \qquad \text{when} \quad j = 2, \ldots, m$$

$$e_{2j} = b_{2j} - a_{2j+1} \qquad \text{when} \quad j = 1, \ldots, m-1$$

$$e_n = b_n$$

and

$$x_k = 1 \qquad \text{when} \quad k = 2j, 2j+1$$

$$x_k = 0 \qquad \text{otherwise}$$

$$y_k = a_{2j+1} \qquad \text{when} \quad k = 2j$$

$$y_k = c_{2j} \qquad \text{when} \quad k = 2j+1$$

$$y_k = 0 \qquad \text{otherwise}$$

that is $x^{(j)} = (0, \ldots, 0, 1, 1, 0, \ldots, 0)^T$, $y^{(j)} = (0, \ldots, 0, a_{2j+1}, c_{2j}, 0, \ldots, 0)^T$.

In matrix notation, A is then written as follows

$$\begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_m \end{pmatrix} + \sum_{j=1}^{m-1} \begin{pmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_m} \end{pmatrix}^{(j)} \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \\ \vdots \\ \mathbf{y_m} \end{pmatrix}^{(j)T}$$

where $n = 2m$.

The 2x2 matrices J_j are immediately invertible, by inspection

i.e.

$$\text{if} \quad J_j = \begin{pmatrix} e_{2j-1} & c_{2j-1} \\ a_{2j} & e_{2j} \end{pmatrix}$$

$$\text{then} \quad J_j^{-1} = \frac{1}{\Delta_j} \begin{pmatrix} e_{2j} & -c_{2j-1} \\ -a_{2j} & e_{2j-1} \end{pmatrix}$$

$$\text{where} \quad \Delta_j = (e_{2j}e_{2j-1} - a_{2j}c_{2j-1})$$

Stage 1

m systems of the form $\begin{pmatrix} u_{2j-1} \\ u_{2j} \end{pmatrix} = J_j^{-1} \begin{pmatrix} d_{2j-1} \\ d_{2j} \end{pmatrix}$ are solved.

Stage 2

m systems of the form $\begin{pmatrix} g_{2j-1,k} \\ g_{2j,k} \end{pmatrix} = J_j^{-1} \begin{pmatrix} x_{2j-1,k} \\ x_{2j,k} \end{pmatrix}$ are solved, where k ranges from 1 to $q-1$ and $q = \log_2 n$.

Stage 3

Matrices $g_{j,k}$ and vector $\mathbf{u}$ are updated using the Sherman-Morrison formula.

The final solution $\mathbf{u}$ is obtained.

Description of variables used

ndim : max problem dimension.

mdim : ndim/2.

qdim : max exponent (to give max problem dimension $ndim = 2^{qdim}$).

n : current problem dimension.

m : n/2.

q : exponent (to give problem dimension $n = 2^q$).

a, b, c : three n-dimensional vectors, storing coefficient matrix A.

u : n-dimensional vector, storing unknown vector (and solution vector, at last step).

d : n-dimensional vector, storing data vector.

e : n-dimensional vector, used to initialise matrices J_j.

g : rectangular matrix of dimension $nx(q-1)$, initialised in stage 2, used and updated in stage 3, in order to update solution vector u.

x : rectangular matrix of dimension $nx(q-1)$, used to factorise coefficient matrix A and to update matrix g and vector u during stage 3 (the variable x is also used to store values of factor array y so that it is no longer necessary to use y).

delta : m-dimensional vector, storing values of Δ_j.

jj1 : array of dimension 2x2xm, storing inverses of matrices J_j.

jjj : 2x2 work matrix, used in stage 1.

uu : work vector of dimension 2, used in stage 1.

dd : as uu.

sol : n-dimensional vector, storing exact solution.

rsol : real value used to calculate exact solution.

rec : real value used to calculate $1/\Delta_j$.

time1, time2 : integers used to calculate elapsed time.

iniz, ifine : integers used in stage 2 to give first, last and step values of do loop indices.

begin, size : integers used in stage 3 to give first and last step of calculation in do loops.

var, go : logical variables, used in stage 3 to signal end-of-work to child processes.

nprocs : number of processors used.

m_set_procs : parallel directive to set number of processors.

m_fork : parallel directive to fork child processes.

Subroutines used

stage3 : to perform calculations required during stage 3 in parallel.

stage33 : to perform last step of stage 3.

partition : returns the updated value of integer variable *begin* to each calling processor, so that a defined part of the total work is assigned to each processor.

dmatvet : to perform matrix/vector multiplication during stage 1.

matvet : to perform matrix/vector multiplication during stage 3.

Description of variables used in subroutines stage3 and stage33

n, m, q : as in main program.

u : as in main program.

g, x : as in main program.

yg : real, work variable used to store the summation of products $x_{j,k}\ g_{j,k}$, as required in the Sherman-Morrison formula.

alfa : real, storing value of $1/(1+yg)$.

u2, u3 : n-dimensional work vectors, used to update vector u.

g2, g3 : n-dimensional work vectors, used to update matrix g.

mgy : square work matrix of dimension nxn, used to store products $alfa\ g_{j,k}\ x_{j,k}$, as required in the Sherman-Morrison formula.

begin, size : as in main program.

var, go : as in main program.

Note

The above specifications and notation apply to all the three Fortran versions of the Recursive Decoupling method (*Parallel.f, Paralleli.f, Parallelo.f*), with the only exception that subroutine *Stage 33* is not used in version *Paralleli.f.*

In the following, the Fortran code for the three mentioned versions is given (related to the first test system).

In addition, we include two more the Fortran listings of version *Parallel.f*, related to the second test system and to the calculation of "reclaimed times" respectively.

In order to calculate the reclaimed times, it has been necessary to parallelize the Pre-stage, Stage 1 and Stage 2 by using the m_fork microtasking routine, instead of using the Doacross parallel directive.

For consistency reasons, all the Fortran listings contains the m_fork construct to perform all the stages required by the algorithm.

Version Parallel.f
(first example)

```
$system
c Parallel.f
c Giulia Spaletta - Dept.of Computer Studies - L.U.T. - Sept.1991
      program decoupling
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      EXTERNAL prestage
      EXTERNAL stage1
      EXTERNAL stage2
      EXTERNAL stage3
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
      COMMON/const4/d
c
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real a(ndim),b(ndim),c(ndim)
      real u(ndim)
      real d(ndim)
      real e(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      real sol(ndim)
      real rsol
c
      integer i,j,k,l
      integer n,m,q
      integer time,time1,time2
      integer iniz,ifine
c
      integer*4 nprocs
      integer*4 m_set_procs
      integer*4 m_fork
c
      integer begin,size
      logical var,go
c
      EQUIVALENCE(sol,e)
c
      open(4,file='Paralleldat',status='new')
c
c
      write(4,*)
      write(4,*)'Program Decoupling (Version Parallel.f -  '
      write(4,*)'                    data file Paralleldat)'
      write(4,*)'Number of used processors is as follows:'
      write(4,*)'    for iterations 1,2,.,q-2     ......    nprocs procs.'
      write(4,*)'    for iteration  q-1           ......    1 processor'
      write(4,*)
c
c
999   continue
c
c reading input data
      write(*,*)
      write(*,*)'exponent q,    where n=2**q   or  n=2*m  ( 2 <= q <= 9 )'
```

```
      read(*,*) q
      n=2**q
      m=n/2
      write(*,*)
      write(*,*)'num of processors '
      write(*,*)'         n.b.   q=2......nprocs=1 '
      write(*,*)'                q=3......nprocs=2 '
      write(*,*)'                q=4......nprocs=4 '
      write(*,*)'                q>=5.....nprocs=8 '
      read(*,*) nprocs
c
c  initialising time variables
      time1=0
      time2=0
      time=0
c
c initialising coefficient matrix A, unknowns vector u, data vector d
      do 2 i=1,n
         b(i)=2.0
         d(i)=0.0
         u(i)=0.0
2     continue
      d(1)=1.0
      do 3 i=1,n
         a(i)=-1.0
         c(i)=-1.0
3     continue
c
c initialising arrays xj, gj    (n.b. yj is not necessary)
      do 4 j=1,q-1
      do 4 i=1,n
         x(i,j)=0.0
         g(i,j)=0.0
4     continue
c
c
c initialising inverses of matrices Jj
      e(1)=b(1)
      e(n)=b(n)
      do 7 j=2,m
         e(2*j-1)=b(2*j-1)-c(2*j-2)
7     continue
      do 8 j=1,m-1
         e(2*j)=b(2*j)-a(2*j+1)
8     continue
c
c
      call _clock_time(time1)
c
c setting number of processors
      i1=m_set_procs(nprocs)
c
c
c
c PRESTAGE:
c calculating delta(j)
c calculating inverses of matrices Jj
c
      call m_fork(prestage)
      call m_kill_procs
c
c
c
c STAGE 1:
c
      call m_fork(stage1)
```

```
      call m_kill_procs
c
c
c
c STAGE 2:
c
      iniz=1
      do 10 k=1,q-1
         ifine=2*iniz
         call m_fork(stage2,k,iniz,ifine)
         iniz=2*iniz
10    continue
      call m_kill_procs
c
c
c
c STAGE 3:
c Number of used processors is as follows:'
c    for iterations 1,2,.,q-2     ......    nprocs procs.
c    for iteration  q-1           ......    1 processor
c
      do 12 k=1,q-2
         begin=1
         go=.true.
         size=2**(k+1)
         var=.true.
             call m_fork(stage3,k)
             call m_sync
12    continue
      call m_kill_procs
c
      k=q-1
      begin=1
      go=.true.
      size=2**(k+1)
      var=.true.
          call stage33(k)
c
2001  continue
      call _clock_time(time2)
c
      time=time2-time1
c
      rsol=real(n+1)
      do 18 l=1,n
         sol(l)=real(n+1-l)/rsol
18    continue
c
      write(4,*)
      write(4,20)(i,sol(i),i=1,n)
20    format(2x,'sol(',i4,'):',f20.10)
c
      write(4,*)
      write(4,30)(i,u(i),i=1,n)
30    format(2x,'u(',i4,'):',f20.10)
c
      do 40 l=1,n
         sol(l)=sol(l)-u(l)
40    continue
c
      write(4,*)
      write(4,60)(i,sol(i),i=1,n)
60    format(2x,'diff(',i4,'):',f20.10)
c
      write(4,*)
      write(4,70)n,m,q
```

```
70    format(2x,'dimension n:',i4,2x,'factor m:',i4,2x,'exponent q:',i4)
c
      write(4,*)
      write(4,80)nprocs
80    format(2x,'number of processors  nprocs:',i4)
c
      write(4,*)
      write(4,90)time/100.0
      write(*,*)
      write(*,90)time/100.0
90    format(2x,'time in sec.:',f20.10)
c
      write(*,*)
      write(4,*)'************************************************'
      write(*,*)
      write(*,*)'continue?   (0=NO, 1=YES)'
      write(*,*)
      read(*,*)num
      if(num.ne.0) go to 999
c
      close(4)
      stop
      end
c
c
c
      subroutine prestage
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
c
      real a(ndim),c(ndim)
      real e(ndim)
      real jj1(2,2,mdim)
c
      real delta(mdim)
      real rec
c
      integer n,m,q
c
c
      do 1 j=1,m
         delta(j)=e(2*j)*e(2*j-1)-a(2*j)*c(2*j-1)
c
         rec=1.0/delta(j)
         jj1(1,1,j)=e(2*j)*rec
         jj1(1,2,j)=-c(2*j-1)*rec
         jj1(2,1,j)=-a(2*j)*rec
         jj1(2,2,j)=e(2*j-1)*rec
1     continue
      return
      end
c
c
c
      subroutine stage1
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const3/jj1
      COMMON/const4/d
```

```
      COMMON/shar20/u
c
      real u(ndim)
      real d(ndim)
      real jj1(2,2,mdim)
      real jjj(2,2),uu(2),dd(2)
c
      integer n,m,q
c
c
      do 10 j=1,m
         dd(1)=d(2*j-1)
         dd(2)=d(2*j)
         jjj(1,1)=jj1(1,1,j)
         jjj(1,2)=jj1(1,2,j)
         jjj(2,1)=jj1(2,1,j)
         jjj(2,2)=jj1(2,2,j)
         call dmatvet(jjj,dd,uu)
         u(2*j-1)=uu(1)
         u(2*j)=uu(2)
10    continue
      return
      end
c
c
c
      subroutine stage2(k,iniz,ifine)
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const3/jj1
c
      COMMON/shar3/g,x
c
      real a(ndim),c(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      integer n,m,q
      integer iniz,ifine
c
c
      do 20 j=iniz,m-iniz,ifine
         g(2*j-1,k)=jj1(1,2,j)
         g(2*j,k)=jj1(2,2,j)
         g(2*j+1,k)=jj1(1,1,j+1)
         g(2*j+2,k)=jj1(2,1,j+1)
         x(2*j,k)=a(2*j+1)
         x(2*j+1,k)=c(2*j)
20    continue
      return
      end
c
c
c
      subroutine stage3(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
```

```
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
            call m_lock()
            call partition(begin)
            call m_unlock()
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
```

```
                do 117 ir=begin,begin+size-1
                   iriga=ir-begin+1
                   g(ir,kk)=g3(iriga)
117             continue
177           continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine stage33(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
            call partition(begin)
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
```

```
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine partition(pbegin)
      COMMON/const1/n,m,q
      COMMON/logi/var,go
      COMMON/misura/size
c
      logical go,var
      integer begin,size,pbegin
c
c
      save begin
            if (var) then
                        begin=1
                        var=.false.
            else
            if (begin.ge.(n-size)) then
                        go=.false.
                 else
                        begin=begin+size
                 endif
            endif
      pbegin=begin
      return
      end
c
c
c
      subroutine dmatvet(a,x,y)
      real a(2,2)
      real x(2),y(2)
      integer i,k
```

```
c
      do 20 i=1,2
         y(i)=0.0
         do 10 k=1,2
            y(i)=y(i)+a(i,k)*x(k)
10       continue
20    continue
c
      return
      end
c
c
c
      subroutine matvet(a,v,c,nn)
      integer ndim
      parameter(ndim=1024)
c
      real a(ndim,nn)
      real v(nn),c(nn)
      real sum
      integer ii,jj
c
      do 10 ii=1,nn
         sum=0.0
         do 9 jj=1,nn
            sum=sum+a(ii,jj)*v(jj)
9        continue
      c(ii)=sum
10    continue
c
      return
      end
c
```

Version Parallel.f
(reclaimed time)

```
$system
c Paralleltime.f
c Giulia Spaletta - Dept.of Computer Studies - L.U.T. - Sept.1991
      program decoupling
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      EXTERNAL prestage
      EXTERNAL stage1
      EXTERNAL stage2
      EXTERNAL stage3
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
      COMMON/const4/d
c
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real a(ndim),b(ndim),c(ndim)
      real u(ndim)
      real d(ndim)
      real e(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      real sol(ndim)
      real rsol
c
      integer i,j,k,l
      integer n,m,q
      integer time,time1(2*qdim),time2(2*qdim),time3(2*qdim)
      integer iniz,ifine
c
      integer*4 nprocs
      integer*4 m_set_procs
      integer*4 m_fork
c
      integer begin,size
      logical var,go
c
      EQUIVALENCE(sol,e)
c
      open(4,file='Paralleltimedat',status='new')
c
c
      write(4,*)
      write(4,*)'Program Decoupling (Version Paralleltime.f -  '
      write(4,*)'                    data file Paralleltimedat)'
      write(4,*)'Number of used processors is as follows:'
      write(4,*)'    for iterations 1,2,.,q-2     ......    nprocs procs.'
      write(4,*)'    for iteration  q-1           ......    1 processor'
      write(4,*)
c
c
999   continue
c
c reading input data
      write(*,*)
      write(*,*)'exponent q,      where n=2**q   or  n=2*m     ( 2 <= q <= 9 )'
```

```
      read(*,*) q
      n=2**q
      m=n/2
      write(*,*)
      write(*,*)'num of processors '
      write(*,*)'         n.b.   q=2......nprocs=1 '
      write(*,*)'                q=3......nprocs=2 '
      write(*,*)'                q=4......nprocs=4 '
      write(*,*)'                q>=5.....nprocs=8 '
      read(*,*) nprocs
c
c  initialising time arrays, in order to obtain reclaimed time
      do 1 k=1,2*q
         time1(k)=0
         time2(k)=0
         time3(k)=0
1     continue
c
c initialising coefficient matrix A, unknowns vector u, data vector d
      do 2 i=1,n
         b(i)=2.0
         d(i)=0.0
         u(i)=0.0
2     continue
      d(1)=1.0
      do 3 i=1,n
         a(i)=-1.0
         c(i)=-1.0
3     continue
c
c initialising arrays xj, gj   (n.b. yj is not necessary)
      do 4 j=1,q-1
      do 4 i=1,n
         x(i,j)=0.0
         g(i,j)=0.0
4     continue
c
c
c initialising inverses of matrices Jj
      e(1)=b(1)
      e(n)=b(n)
      do 7 j=2,m
         e(2*j-1)=b(2*j-1)-c(2*j-2)
7     continue
      do 8 j=1,m-1
         e(2*j)=b(2*j)-a(2*j+1)
8     continue
c
c
c
c setting number of processors
      il=m_set_procs(nprocs)
c
c
c
c PRESTAGE:
c calculating delta(j)
c calculating inverses of matrices Jj
c
      call m_fork(prestage,time1(1))
      call _clock_time(time2(1))
      call m_kill_procs
c
c
c
c STAGE 1:
```

```
c
      call m_fork(stage1,time1(2))
      call _clock_time(time2(2))
      call m_kill_procs
c
c
c
c STAGE 2:
c
      iniz=1
      do 10 k=1,q-1
         ifine=2*iniz
         call m_fork(stage2,k,iniz,ifine,time1(k+2))
         call _clock_time(time2(k+2))
      iniz=2*iniz
10    continue
      call m_kill_procs
c
c
c
c STAGE 3:
c Number of used processors is as follows:'
c    for iterations 1,2,.,q-2     ......    nprocs procs.
c    for iteration  q-1           ......    1 processor
c
      do 12 k=1,q-2
         begin=1
         go=.true.
         size=2**(k+1)
         var=.true.
              call m_fork(stage3,k,time1(k+q+1))
              call _clock_time(time2(k+q+1))
              call m_sync
12    continue
      call m_kill_procs
c
      k=q-1
      begin=1
      go=.true.
      size=2**(k+1)
      var=.true.
           call _clock_time(time1(k+q+1))
           call stage33(k)
c
2001  continue
      call _clock_time(time2(k+q+1))
c
c
      rsol=real(n+1)
      do 18 l=1,n
         sol(l)=real(n+1-l)/rsol
18    continue
c
      do 19 k=1,2*q
         time3(k)=time2(k)-time1(k)
19    continue
      time=0
      do 20 k=1,2*q
         time=time+time3(k)
20    continue
c
c
      write(4,*)
      write(4,21)(i,sol(i),i=1,n)
21    format(2x,'sol(',i4,'):',f20.10)
c
```

```
      write(4,*)
      write(4,30)(i,u(i),i=1,n)
30    format(2x,'u(',i4,'):',f20.10)
c
      do 40 l=1,n
         sol(l)=sol(l)-u(l)
40    continue
      write(4,50)(i,sol(i),i=1,n)
50    format(2x,'diff(',i4,'):',f20.10)
c
      write(4,*)
      write(4,60)n,m,q
60    format(2x,'dimension n:',i4,2x,'factor m:',i4,2x,'exponent q:',i4)
      write(4,*)
      write(4,70)nprocs
70    format(2x,'number of processors  nprocs:',i4)
c
      write(4,*)
      write(4,80)time/100.0
      write(*,*)
      write(*,80)time/100.0
80    format(2x,'reclaimed time in sec.:',f20.10)
c
      write(*,*)
      write(4,*)'************************************************'
      write(*,*)
      write(*,*)'continue?   (0=NO, 1=YES)'
      write(*,*)
      read(*,*)num
      if(num.ne.0) go to 999
c
      close(4)
      stop
      end
c
c
c
      subroutine prestage(time1)
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
c
      real a(ndim),c(ndim)
      real e(ndim)
      real jj1(2,2,mdim)
c
      real delta(mdim)
      real rec
c
      integer n,m,q
      integer j,time1
c
      call m_single()
           call _clock_time(time1)
      call m_multi()
c
      do 1 j=1,m
         delta(j)=e(2*j)*e(2*j-1)-a(2*j)*c(2*j-1)
c
         rec=1.0/delta(j)
         jj1(1,1,j)=e(2*j)*rec
         jj1(1,2,j)=-c(2*j-1)*rec
```

```
         jj1(2,1,j)=-a(2*j)*rec
         jj1(2,2,j)=e(2*j-1)*rec
1     continue
      return
      end
c
c
c
      subroutine stage1(time1)
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const3/jj1
      COMMON/const4/d
      COMMON/shar20/u
c
      real u(ndim)
      real d(ndim)
      real jj1(2,2,mdim)
      real jjj(2,2),uu(2),dd(2)
c
      integer n,m,q
      integer time1,j
c
      call m_single()
           call _clock_time(time1)
      call m_multi()
c
      do 10 j=1,m
         dd(1)=d(2*j-1)
         dd(2)=d(2*j)
         jjj(1,1)=jj1(1,1,j)
         jjj(1,2)=jj1(1,2,j)
         jjj(2,1)=jj1(2,1,j)
         jjj(2,2)=jj1(2,2,j)
         call dmatvet(jjj,dd,uu)
         u(2*j-1)=uu(1)
         u(2*j)=uu(2)
10    continue
      return
      end
c
c
c
      subroutine stage2(k,iniz,ifine,time1)
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const3/jj1
c
      COMMON/shar3/g,x
c
      real a(ndim),c(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      integer n,m,q
      integer time1
      integer iniz,ifine
c
      call m_single()
           call _clock_time(time1)
```

```
      call m_multi()
c
      do 20 j=iniz,m-iniz,ifine
         g(2*j-1,k)=jj1(1,2,j)
         g(2*j,k)=jj1(2,2,j)
         g(2*j+1,k)=jj1(1,1,j+1)
         g(2*j+2,k)=jj1(2,1,j+1)
         x(2*j,k)=a(2*j+1)
         x(2*j+1,k)=c(2*j)
20    continue
      return
      end
c
c
c
      subroutine stage3(k,time1)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
      integer time1
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
      call m_single()
           call _clock_time(time1)
      call m_multi()
c
888   continue
c
      call m_lock()
           call partition(begin)
      call m_unlock()
c
      if (go) then
c
             yg=0.0
             do 14 ir=begin,begin+size-1
                yg=yg+x(ir,k)*g(ir,k)
14           continue
             alfa=1.0/(1.0+yg)
c
             do 15 ir=begin,begin+size-1
                iriga=ir-begin+1
             do 15 ic=begin,begin+size-1
                icol=ic-begin+1
```

```
                mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
                if (ir.eq.ic) then
                    mgy(iriga,icol)=1.0-mgy(iriga,icol)
                else
                    mgy(iriga,icol)=-mgy(iriga,icol)
                endif
15            continue
c
              do 16 ir=begin,begin+size-1
                 iriga=ir-begin+1
                 u2(iriga)=u(ir)
16            continue
c
              call matvet(mgy,u2,u3,size)
c
              do 116 ir=begin,begin+size-1
                 iriga=ir-begin+1
                 u(ir)=u3(iriga)
116           continue
c
              do 177 kk=k+1,q-1
                 do 17 ir=begin,begin+size-1
                    iriga=ir-begin+1
                    g2(iriga)=g(ir,kk)
17               continue
c
              call matvet(mgy,g2,g3,size)
c
                 do 117 ir=begin,begin+size-1
                    iriga=ir-begin+1
                    g(ir,kk)=g3(iriga)
117              continue
177           continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine stage33(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
```

```
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
c
      call partition(begin)
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine partition(pbegin)
```

```
      COMMON/const1/n,m,q
      COMMON/logi/var,go
      COMMON/misura/size
c
      logical go,var
      integer begin,size,pbegin
c
c
      save begin
             if (var) then
                        begin=1
                        var=.false.
             else
             if (begin.ge.(n-size)) then
                        go=.false.
                 else
                        begin=begin+size
                 endif
             endif
      pbegin=begin
      return
      end
c
c
c
      subroutine dmatvet(a,x,y)
      real a(2,2)
      real x(2),y(2)
      integer i,k
c
      do 20 i=1,2
         y(i)=0.0
         do 10 k=1,2
            y(i)=y(i)+a(i,k)*x(k)
10       continue
20    continue
c
      return
      end
c
c
c
      subroutine matvet(a,v,c,nn)
      integer ndim
      parameter(ndim=1024)
c
      real a(ndim,nn)
      real v(nn),c(nn)
      real sum
      integer ii,jj
c
      do 10 ii=1,nn
         sum=0.0
         do 9 jj=1,nn
            sum=sum+a(ii,jj)*v(jj)
9        continue
      c(ii)=sum
10    continue
c
      return
      end
c
```

Version Parallel.f
(second example)

```
$system
c Parallel2.f
c Giulia Spaletta - Dept.of Computer Studies - L.U.T. - Sept.1990
      program decoupling
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      EXTERNAL prestage
      EXTERNAL stage1
      EXTERNAL stage2
      EXTERNAL stage3
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
      COMMON/const4/d
c
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real a(ndim),b(ndim),c(ndim)
      real u(ndim)
      real d(ndim)
      real e(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
c
      integer i,j,k,l
      integer n,m,q
      integer time,time1,time2
      integer iniz,ifine
c
      integer*4 nprocs
      integer*4 m_set_procs
      integer*4 m_fork
c
      integer begin,size
      logical var,go
c
c
      open(4,file='Parallel2dat',status='new')
c
c
      write(4,*)
      write(4,*)'Program Decoupling (Version Parallel2.f -  '
      write(4,*)'                    data file Parallel2dat)'
      write(4,*)'Number of used processors is as follows:'
      write(4,*)'    for iterations 1,2,.,q-2     ......    nprocs procs.'
      write(4,*)'    for iteration  q-1           ......    1 processor'
      write(4,*)
c
c
999   continue
c
c reading input data
      write(*,*)
      write(*,*)'exponent q,      where n=2**q   or  n=2*m     ( 2 <= q <= 9 )'
      read(*,*) q
      n=2**q
      m=n/2
```

```
      write(*,*)
      write(*,*)'num of processors '
      write(*,*)'          n.b.    q=2......nprocs=1 '
      write(*,*)'                  q=3......nprocs=2 '
      write(*,*)'                  q=4......nprocs=4 '
      write(*,*)'                  q>=5.....nprocs=8 '
      read(*,*) nprocs
c
c  initialising time variables
      time1=0
      time2=0
      time=0
c
c initialising coefficient matrix A, unknowns vector u, data vector d
      do 2 i=1,n
         u(i)=0.0
2     continue
      do 28 i=1,n-1,8
         a(i)=0.0
         a(i+1)=3.0
         a(i+2)=2.0
         a(i+3)=2.0
         a(i+4)=1.0
         a(i+5)=4.0
         a(i+6)=7.0
         a(i+7)=1.0
c
         b(i)=2.0
         b(i+1)=5.0
         b(i+2)=3.0
         b(i+3)=4.0
         b(i+4)=4.0
         b(i+5)=6.0
         b(i+6)=8.0
         b(i+7)=3.0
c
         c(i)=1.0
         c(i+1)=2.0
         c(i+2)=1.0
         c(i+3)=1.0
         c(i+4)=3.0
         c(i+5)=1.0
         c(i+6)=1.0
         c(i+7)=0.0
c
         d(i)=1.0
         d(i+1)=0.0
         d(i+2)=0.0
         d(i+3)=1.0
         d(i+4)=0.0
         d(i+5)=1.0
         d(i+6)=0.0
         d(i+7)=2.0
28    continue
      do 3 i=1,n
         a(i)=-a(i)
         c(i)=-c(i)
3     continue
c
c initialising arrays xj, gj   (n.b. yj is not necessary)
      do 4 j=1,q-1
      do 4 i=1,n
         x(i,j)=0.0
         g(i,j)=0.0
4     continue
c
```

```
c
c initialising inverses of matrices Jj
      e(1)=b(1)
      e(n)=b(n)
      do 7 j=2,m
         e(2*j-1)=b(2*j-1)-c(2*j-2)
7     continue
      do 8 j=1,m-1
         e(2*j)=b(2*j)-a(2*j+1)
8     continue
c
c
      call _clock_time(time1)
c
c setting number of processors
      i1=m_set_procs(nprocs)
c
c
c
c PRESTAGE:
c calculating delta(j)
c calculating inverses of matrices Jj
c
      call m_fork(prestage)
      call m_kill_procs
c
c
c
c STAGE 1:
c
      call m_fork(stage1)
      call m_kill_procs
c
c
c
c STAGE 2:
c
      iniz=1
      do 10 k=1,q-1
         ifine=2*iniz
         call m_fork(stage2,k,iniz,ifine)
         iniz=2*iniz
10    continue
      call m_kill_procs
c
c
c
c STAGE 3:
c Number of used processors is as follows:'
c    for iterations 1,2,.,q-2     ......    nprocs procs.
c    for iteration  q-1           ......    1 processor
c
      do 12 k=1,q-2
         begin=1
         go=.true.
         size=2**(k+1)
         var=.true.
             call m_fork(stage3,k)
             call m_sync
12    continue
      call m_kill_procs
c
      k=q-1
      begin=1
      go=.true.
      size=2**(k+1)
```

```
      var=.true.
          call stage33(k)
c
2001  continue
      call _clock_time(time2)
c
      time=time2-time1
c
      write(4,*)
      write(4,70)(i,u(i),i=1,n)
70    format(2x,'u(',i4,'):',f20.10)
c
      write(4,*)
      write(4,72)n,m,q
72    format(2x,'dimension n:',i4,2x,'factor m:',i4,2x,'exponent q:',i4)
c
      write(4,*)
      write(4,73)nprocs
73    format(2x,'number of processors  nprocs:',i4)
c
      write(4,*)
      write(4,74)time/100.0
      write(*,*)
      write(*,74)time/100.0
74    format(2x,'time in sec.:',f20.10)
c
      write(*,*)
      write(4,*)'************************************************'
      write(*,*)
      write(*,*)'continue?   (0=NO, 1=YES)'
      write(*,*)
      read(*,*)num
      if(num.ne.0) go to 999
c
      close(4)
      stop
      end
c
c
c
      subroutine prestage
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
c
      real a(ndim),c(ndim)
      real e(ndim)
      real jj1(2,2,mdim)
c
      real delta(mdim)
      real rec
c
      integer n,m,q
c
c
      do 1 j=1,m
         delta(j)=e(2*j)*e(2*j-1)-a(2*j)*c(2*j-1)
c
         rec=1.0/delta(j)
         jj1(1,1,j)=e(2*j)*rec
         jj1(1,2,j)=-c(2*j-1)*rec
         jj1(2,1,j)=-a(2*j)*rec
```

```
         jj1(2,2,j)=e(2*j-1)*rec
1     continue
      return
      end
c
c
c
      subroutine stage1
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const3/jj1
      COMMON/const4/d
      COMMON/shar20/u
c
      real u(ndim)
      real d(ndim)
      real jj1(2,2,mdim)
      real jjj(2,2),uu(2),dd(2)
c
      integer n,m,q
c
c
      do 10 j=1,m
         dd(1)=d(2*j-1)
         dd(2)=d(2*j)
         jjj(1,1)=jj1(1,1,j)
         jjj(1,2)=jj1(1,2,j)
         jjj(2,1)=jj1(2,1,j)
         jjj(2,2)=jj1(2,2,j)
         call dmatvet(jjj,dd,uu)
         u(2*j-1)=uu(1)
         u(2*j)=uu(2)
10    continue
      return
      end
c
c
c
      subroutine stage2(k,iniz,ifine)
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const3/jj1
c
      COMMON/shar3/g,x
c
      real a(ndim),c(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      integer n,m,q
      integer iniz,ifine
c
c
      do 20 j=iniz,m-iniz,ifine
         g(2*j-1,k)=jj1(1,2,j)
         g(2*j,k)=jj1(2,2,j)
         g(2*j+1,k)=jj1(1,1,j+1)
         g(2*j+2,k)=jj1(2,1,j+1)
         x(2*j,k)=a(2*j+1)
         x(2*j+1,k)=c(2*j)
```

```
20    continue
      return
      end
c
c
c
      subroutine stage3(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
      call m_lock()
           call partition(begin)
      call m_unlock()
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
```

```
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine stage33(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
      call partition(begin)
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
```

```
14            continue
              alfa=1.0/(1.0+yg)
c
              do 15 ir=begin,begin+size-1
                 iriga=ir-begin+1
              do 15 ic=begin,begin+size-1
                 icol=ic-begin+1
                 mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
                 if (ir.eq.ic) then
                     mgy(iriga,icol)=1.0-mgy(iriga,icol)
                 else
                     mgy(iriga,icol)=-mgy(iriga,icol)
                 endif
15            continue
c
              do 16 ir=begin,begin+size-1
                 iriga=ir-begin+1
                 u2(iriga)=u(ir)
16            continue
c
              call matvet(mgy,u2,u3,size)
c
              do 116 ir=begin,begin+size-1
                 iriga=ir-begin+1
                 u(ir)=u3(iriga)
116           continue
c
              do 177 kk=k+1,q-1
                 do 17 ir=begin,begin+size-1
                    iriga=ir-begin+1
                    g2(iriga)=g(ir,kk)
17               continue
c
              call matvet(mgy,g2,g3,size)
c
                 do 117 ir=begin,begin+size-1
                    iriga=ir-begin+1
                    g(ir,kk)=g3(iriga)
117              continue
177           continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine partition(pbegin)
      COMMON/const1/n,m,q
      COMMON/logi/var,go
      COMMON/misura/size
c
      logical go,var
      integer begin,size,pbegin
c
c
      save begin
             if (var) then
                       begin=1
                       var=.false.
             else
             if (begin.ge.(n-size)) then
                       go=.false.
```

```
                else
                        begin=begin+size
                endif
            endif
        pbegin=begin
        return
        end
c
c
c
        subroutine dmatvet(a,x,y)
        real a(2,2)
        real x(2),y(2)
        integer i,k
c
        do 20 i=1,2
           y(i)=0.0
           do 10 k=1,2
              y(i)=y(i)+a(i,k)*x(k)
10         continue
20      continue
c
        return
        end
c
c
c
        subroutine matvet(a,v,c,nn)
        integer ndim
        parameter(ndim=1024)
c
        real a(ndim,nn)
        real v(nn),c(nn)
        real sum
        integer ii,jj
c
        do 10 ii=1,nn
           sum=0.0
           do 9 jj=1,nn
              sum=sum+a(ii,jj)*v(jj)
9          continue
        c(ii)=sum
10      continue
c
        return
        end
c
```

Version Paralleli.f
(first example)

```
$system
c Paralleli.f
c Giulia Spaletta - Dept.of Computer Studies - L.U.T. - Sept.1990
      program decoupling
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      EXTERNAL prestage
      EXTERNAL stage1
      EXTERNAL stage2
      EXTERNAL stage3
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
      COMMON/const4/d
c
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real a(ndim),b(ndim),c(ndim)
      real u(ndim)
      real d(ndim)
      real e(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      real sol(ndim)
      real rsol
c
      integer i,j,k,l
      integer n,m,q
      integer time,time1,time2
      integer iniz,ifine
c
      integer*4 nprocs
      integer*4 m_set_procs
      integer*4 m_fork
c
      integer begin,size
      logical var,go
c
      EQUIVALENCE(sol,e)
c
      open(4,file='Parallelidat',status='new')
c
c
      write(4,*)
      write(4,*)'Program Decoupling (Version Paralleli.f -  '
      write(4,*)'                    data file Parallelidat)'
      write(4,*)'Number of used processors is as follows:'
      write(4,*)'    for iterations 1,2,.,q-1     ......    nprocs procs.'
      write(4,*)
c
c
999   continue
c
c reading input data
      write(*,*)
      write(*,*)'exponent q,      where n=2**q   or  n=2*m     ( 2 <= q <= 9 )'
      read(*,*) q
```

```
      n=2**q
      m=n/2
      write(*,*)
      write(*,*)'num of processors '
      write(*,*)'         n.b.   q=2.....nprocs=1 '
      write(*,*)'                q=3.....nprocs=2 '
      write(*,*)'                q=4.....nprocs=4 '
      write(*,*)'                q>=5.....nprocs=8 '
      read(*,*) nprocs
c
c  initialising time variables
      time1=0
      time2=0
      time=0
c
c initialising coefficient matrix A, unknowns vector u, data vector d
      do 2 i=1,n
         b(i)=2.0
         d(i)=0.0
         u(i)=0.0
2     continue
      d(1)=1.0
      do 3 i=1,n
         a(i)=-1.0
         c(i)=-1.0
3     continue
c
c initialising arrays xj, gj   (n.b. yj is not necessary)
      do 4 j=1,q-1
      do 4 i=1,n
         x(i,j)=0.0
         g(i,j)=0.0
4     continue
c
c
c initialising inverses of matrices Jj
      e(1)=b(1)
      e(n)=b(n)
      do 7 j=2,m
         e(2*j-1)=b(2*j-1)-c(2*j-2)
7     continue
      do 8 j=1,m-1
         e(2*j)=b(2*j)-a(2*j+1)
8     continue
c
c
      call _clock_time(time1)
c
c setting number of processors
      i1=m_set_procs(nprocs)
c
c
c
c PRESTAGE:
c calculating delta(j)
c calculating inverses of matrices Jj
c
      call m_fork(prestage)
      call m_kill_procs
c
c
c
c
c STAGE 1:
c
      call m_fork(stage1)
```

```
      call m_kill_procs
c
c
c
c STAGE 2:
c
      iniz=1
      do 10 k=1,q-1
         ifine=2*iniz
         call m_fork(stage2,k,iniz,ifine)
         iniz=2*iniz
10    continue
      call m_kill_procs
c
c
c
c STAGE 3:
c Number of used processors is as follows:'
c    for iterations 1,2,.,q-1     ......    nprocs procs.
c
      do 12 k=1,q-1
         begin=1
         go=.true.
         size=2**(k+1)
         var=.true.
             call m_fork(stage3,k)
             call m_sync
12    continue
      call m_kill_procs
c
c
2001  continue
      call _clock_time(time2)
c
      time=time2-time1
c
      rsol=real(n+1)
      do 18 l=1,n
         sol(l)=real(n+1-l)/rsol
18    continue
c
      write(4,*)
      write(4,20)(i,sol(i),i=1,n)
20    format(2x,'sol(',i4,'):',f20.10)
c
      write(4,*)
      write(4,30)(i,u(i),i=1,n)
30    format(2x,'u(',i4,'):',f20.10)
c
      do 40 l=1,n
         sol(l)=sol(l)-u(l)
40    continue
c
      write(4,*)
      write(4,50)(i,sol(i),i=1,n)
50    format(2x,'diff(',i4,'):',f20.10)
c
      write(4,*)
      write(4,60)n,m,q
60    format(2x,'dimension n:',i4,2x,'factor m:',i4,2x,'exponent q:',i4)
c
      write(4,*)
      write(4,70)nprocs
70    format(2x,'number of processors  nprocs:',i4)
c
      write(4,*)
```

```
      write(4,80)time/100.0
      write(*,*)
      write(*,80)time/100.0
80    format(2x,'time in sec.:',f20.10)
c
      write(*,*)
      write(4,*)'************************************************'
      write(*,*)
      write(*,*)'continue?   (0=NO, 1=YES)'
      write(*,*)
      read(*,*)num
      if(num.ne.0) go to 999
c
      close(4)
      stop
      end
c
c
c
      subroutine prestage
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
c
      real a(ndim),c(ndim)
      real e(ndim)
      real jj1(2,2,mdim)
c
      real delta(mdim)
      real rec
c
      integer n,m,q
c
c
      do 1 j=1,m
         delta(j)=e(2*j)*e(2*j-1)-a(2*j)*c(2*j-1)
c
         rec=1.0/delta(j)
         jj1(1,1,j)=e(2*j)*rec
         jj1(1,2,j)=-c(2*j-1)*rec
         jj1(2,1,j)=-a(2*j)*rec
         jj1(2,2,j)=e(2*j-1)*rec
1     continue
      return
      end
c
c
c
      subroutine stage1
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const3/jj1
      COMMON/const4/d
      COMMON/shar20/u
c
      real u(ndim)
      real d(ndim)
      real jj1(2,2,mdim)
      real jjj(2,2),uu(2),dd(2)
c
```

```
      integer n,m,q
c
c
      do 10 j=1,m
         dd(1)=d(2*j-1)
         dd(2)=d(2*j)
         jjj(1,1)=jj1(1,1,j)
         jjj(1,2)=jj1(1,2,j)
         jjj(2,1)=jj1(2,1,j)
         jjj(2,2)=jj1(2,2,j)
         call dmatvet(jjj,dd,uu)
         u(2*j-1)=uu(1)
         u(2*j)=uu(2)
10    continue
      return
      end
c
c
c
      subroutine stage2(k,iniz,ifine)
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const3/jj1
c
      COMMON/shar3/g,x
c
      real a(ndim),c(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      integer n,m,q
      integer iniz,ifine
c
c
      do 20 j=iniz,m-iniz,ifine
         g(2*j-1,k)=jj1(1,2,j)
         g(2*j,k)=jj1(2,2,j)
         g(2*j+1,k)=jj1(1,1,j+1)
         g(2*j+2,k)=jj1(2,1,j+1)
         x(2*j,k)=a(2*j+1)
         x(2*j+1,k)=c(2*j)
20    continue
      return
      end
c
c
c
      subroutine stage3(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
```

```
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
      call m_lock()
           call partition(begin)
      call m_unlock()
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
```

```
            return
        endif
        go to 888
c
        end
c
c
c
        subroutine partition(pbegin)
        COMMON/const1/n,m,q
        COMMON/logi/var,go
        COMMON/misura/size
c
        logical go,var
        integer begin,size,pbegin
c
c
        save begin
               if (var) then
                         begin=1
                         var=.false.
               else
               if (begin.ge.(n-size)) then
                         go=.false.
                   else
                         begin=begin+size
                   endif
               endif
        pbegin=begin
        return
        end
c
c
c
        subroutine dmatvet(a,x,y)
        real a(2,2)
        real x(2),y(2)
        integer i,k
c
        do 20 i=1,2
           y(i)=0.0
           do 10 k=1,2
              y(i)=y(i)+a(i,k)*x(k)
10         continue
20      continue
c
        return
        end
c
c
c
        subroutine matvet(a,v,c,nn)
        integer ndim
        parameter(ndim=1024)
c
        real a(ndim,nn)
        real v(nn),c(nn)
        real sum
        integer ii,jj
c
        do 10 ii=1,nn
           sum=0.0
           do 9 jj=1,nn
              sum=sum+a(ii,jj)*v(jj)
9          continue
        c(ii)=sum
```

```
10      continue
c
        return
        end
c
```

Version Parallelo.f
(first example)

```
$system
c Parallelo.f
c Giulia Spaletta - Dept.of Computer Studies - L.U.T. - Sept.1991
      program decoupling
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      EXTERNAL prestage
      EXTERNAL stage1
      EXTERNAL stage2
      EXTERNAL stage3
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
      COMMON/const4/d
c
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real a(ndim),b(ndim),c(ndim)
      real u(ndim)
      real d(ndim)
      real e(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      real sol(ndim)
      real rsol
c
      integer i,j,k,l
      integer n,m,q
      integer time,time1,time2
      integer iniz,ifine
c
      integer*4 nprocs
      integer*4 m_set_procs
      integer*4 m_fork
c
      integer begin,size
      logical var,go
c
      EQUIVALENCE(sol,e)
c
      open(4,file='Parallelodat',status='new')
c
c
      write(4,*)
      write(4,*)'Program Decoupling (Version Parallelo.f -  '
      write(4,*)'                    data file Parallelodat)'
      write(4,*)'Number of used processors is as follows:'
      write(4,*)'    for iterations 1,2,.,q-4     ......    nprocs procs.'
      write(4,*)'    for iteration  q-3           ......    4 processor'
      write(4,*)'    for iteration  q-2           ......    2 processor'
      write(4,*)'    for iteration  q-1           ......    1 processor'
      write(4,*)
c
c
999   continue
c
c reading input data
```

```
      write(*,*)
      write(*,*)
      write(*,*)'exponent q,      where n=2**q   or  n=2*m     ( 2 <= q <= 9 )'
      read(*,*) q
      n=2**q
      m=n/2
      write(*,*)
      write(*,*)
      write(*,*)'num of processors '
      write(*,*)'         n.b.   q=2......nprocs=1 '
      write(*,*)'                q=3......nprocs=2 '
      write(*,*)'                q=4......nprocs=4 '
      write(*,*)'                q>=5.....nprocs=8 '
      read(*,*) nprocs
c
c  initialising time variables
      time1=0
      time2=0
      time=0
c
c initialising coefficient matrix A, unknowns vector u, data vector d
      do 2 i=1,n
         b(i)=2.0
         d(i)=0.0
         u(i)=0.0
2     continue
      d(1)=1.0
      do 3 i=1,n
         a(i)=-1.0
         c(i)=-1.0
3     continue
c
c initialising arrays xj, gj   (n.b. yj is not necessary)
      do 4 j=1,q-1
      do 4 i=1,n
         x(i,j)=0.0
         g(i,j)=0.0
4     continue
c
c
c initialising inverses of matrices Jj
      e(1)=b(1)
      e(n)=b(n)
      do 7 j=2,m
         e(2*j-1)=b(2*j-1)-c(2*j-2)
7     continue
      do 8 j=1,m-1
         e(2*j)=b(2*j)-a(2*j+1)
8     continue
c
c
      call _clock_time(time1)
c
c setting number of processors
      i1=m_set_procs(nprocs)
c
c
c
c PRESTAGE:
c calculating delta(j)
c calculating inverses of matrices Jj
c
      call m_fork(prestage)
      call m_kill_procs
c
c
```

```
c
c STAGE 1:
c
      call m_fork(stage1)
      call m_kill_procs
c
c
c
c STAGE 2:
c
      iniz=1
      do 10 k=1,q-1
         ifine=2*iniz
         call m_fork(stage2,k,iniz,ifine)
         iniz=2*iniz
10    continue
      call m_kill_procs
c
c
c
c STAGE 3:
c Number of used processors is as follows:'
c    for iterations 1,2,.,q-4     ......    nprocs procs.
c    for iteration  q-3           ......    nprocs processors
c    for iteration  q-2           ......    nprocs processors
c    for iteration  q-1           ......    1 processor
c
      if (q.le.4) go to 303
      do 12 k=1,q-4
         begin=1
         go=.true.
         size=2**(k+1)
         var=.true.
             call m_fork(stage3,k)
             call m_sync
12    continue
      call m_kill_procs
c
303   continue
      if (q.le.3) go to 302
      k=q-3
      il=m_set_procs(4)
      begin=1
      go=.true.
      size=2**(k+1)
      var=.true.
          call m_fork(stage3,k)
      call m_kill_procs
c
302   continue
      if (q.le.2) go to 301
      k=q-2
      il=m_set_procs(2)
      begin=1
      go=.true.
      size=2**(k+1)
      var=.true.
          call m_fork(stage3,k)
      call m_kill_procs
c
301   continue
      k=q-1
      begin=1
      go=.true.
      size=2**(k+1)
      var=.true.
```

```
          call stage33(k)
c
2001  continue
      call _clock_time(time2)
c
      time=time2-time1
c
      rsol=real(n+1)
      do 18 l=1,n
         sol(l)=real(n+1-l)/rsol
18    continue
c
      write(4,*)
      write(4,20)(i,sol(i),i=1,n)
20    format(2x,'sol(',i4,'):',f20.10)
c
      write(4,*)
      write(4,30)(i,u(i),i=1,n)
30    format(2x,'u(',i4,'):',f20.10)
c
      do 40 l=1,n
         sol(l)=sol(l)-u(l)
40    continue
c
      write(4,*)
      write(4,50)(i,sol(i),i=1,n)
50    format(2x,'diff(',i4,'):',f20.10)
c
      write(4,*)
      write(4,60)n,m,q
60    format(2x,'dimension n:',i4,2x,'factor m:',i4,2x,'exponent q:',i4)
c
      write(4,*)
      write(4,70)nprocs
70    format(2x,'number of processors  nprocs:',i4)
c
      write(4,*)
      write(4,80)time/100.0
      write(*,*)
      write(*,80)time/100.0
80    format(2x,'time in sec.:',f20.10)
c
      write(*,*)
      write(4,*)'************************************************'
      write(*,*)
      write(*,*)'continue?   (0=NO, 1=YES)'
      write(*,*)
      read(*,*)num
      if(num.ne.0) go to 999
c
      close(4)
      stop
      end
c
c
c
      subroutine prestage
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const22/e
      COMMON/const3/jj1
c
      real a(ndim),c(ndim)
```

```
      real e(ndim)
      real jj1(2,2,mdim)
c
      real delta(mdim)
      real rec
c
      integer n,m,q
c
c
      do 1 j=1,m
         delta(j)=e(2*j)*e(2*j-1)-a(2*j)*c(2*j-1)
c
         rec=1.0/delta(j)
         jj1(1,1,j)=e(2*j)*rec
         jj1(1,2,j)=-c(2*j-1)*rec
         jj1(2,1,j)=-a(2*j)*rec
         jj1(2,2,j)=e(2*j-1)*rec
1     continue
      return
      end
c
c
c
      subroutine stage1
      integer ndim,mdim
      parameter(ndim=1024,mdim=512)
c
      COMMON/const1/n,m,q
      COMMON/const3/jj1
      COMMON/const4/d
      COMMON/shar20/u
c
      real u(ndim)
      real d(ndim)
      real jj1(2,2,mdim)
      real jjj(2,2),uu(2),dd(2)
c
      integer n,m,q
c
c
      do 10 j=1,m
         dd(1)=d(2*j-1)
         dd(2)=d(2*j)
         jjj(1,1)=jj1(1,1,j)
         jjj(1,2)=jj1(1,2,j)
         jjj(2,1)=jj1(2,1,j)
         jjj(2,2)=jj1(2,2,j)
         call dmatvet(jjj,dd,uu)
         u(2*j-1)=uu(1)
         u(2*j)=uu(2)
10    continue
      return
      end
c
c
c
      subroutine stage2(k,iniz,ifine)
      integer ndim,mdim,qdim
      parameter(ndim=1024,mdim=512,qdim=10)
c
      COMMON/const1/n,m,q
      COMMON/const2/a,c
      COMMON/const3/jj1
c
      COMMON/shar3/g,x
c
```

```
      real a(ndim),c(ndim)
      real g(ndim,qdim-1)
      real x(ndim,qdim-1)
      real jj1(2,2,mdim)
c
      integer n,m,q
      integer iniz,ifine
c
c
      do 20 j=iniz,m-iniz,ifine
         g(2*j-1,k)=jj1(1,2,j)
         g(2*j,k)=jj1(2,2,j)
         g(2*j+1,k)=jj1(1,1,j+1)
         g(2*j+2,k)=jj1(2,1,j+1)
         x(2*j,k)=a(2*j+1)
         x(2*j+1,k)=c(2*j)
20    continue
      return
      end
c
c
c
      subroutine stage3(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
      call m_lock()
           call partition(begin)
      call m_unlock()
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
```

```
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
      subroutine stage33(k)
      integer nndim,mmdim,qqdim
      parameter(nndim=1024,mmdim=512,qqdim=10)
c
      COMMON/const1/n,m,q
      COMMON/shar20/u
      COMMON/shar3/g,x
c
      COMMON/logi/var,go
      COMMON/misura/size
c
      real u(nndim)
      real g(nndim,qqdim-1)
      real x(nndim,qqdim-1)
      real alfa,yg
      real u2(nndim),u3(nndim),g2(nndim),g3(nndim),g4(nndim)
      real mgy(nndim,nndim)
c
      integer ir,ic,kk
      integer n,m,q,k
      integer iriga,icol
```

```
c
      integer begin,size
      logical go,var
c
      EQUIVALENCE(u2,g2)
      EQUIVALENCE(u3,g3)
c
c
888   continue
      call partition(begin)
c
      if (go) then
c
            yg=0.0
            do 14 ir=begin,begin+size-1
               yg=yg+x(ir,k)*g(ir,k)
14          continue
            alfa=1.0/(1.0+yg)
c
            do 15 ir=begin,begin+size-1
               iriga=ir-begin+1
            do 15 ic=begin,begin+size-1
               icol=ic-begin+1
               mgy(iriga,icol)=g(ir,k)*x(ic,k)*alfa
               if (ir.eq.ic) then
                   mgy(iriga,icol)=1.0-mgy(iriga,icol)
               else
                   mgy(iriga,icol)=-mgy(iriga,icol)
               endif
15          continue
c
            do 16 ir=begin,begin+size-1
               iriga=ir-begin+1
               u2(iriga)=u(ir)
16          continue
c
            call matvet(mgy,u2,u3,size)
c
            do 116 ir=begin,begin+size-1
               iriga=ir-begin+1
               u(ir)=u3(iriga)
116         continue
c
            do 177 kk=k+1,q-1
               do 17 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g2(iriga)=g(ir,kk)
17             continue
c
            call matvet(mgy,g2,g3,size)
c
               do 117 ir=begin,begin+size-1
                  iriga=ir-begin+1
                  g(ir,kk)=g3(iriga)
117            continue
177         continue
c
      else
            return
      endif
      go to 888
c
      end
c
c
c
```

```
      subroutine partition(pbegin)
      COMMON/const1/n,m,q
      COMMON/logi/var,go
      COMMON/misura/size
c
      logical go,var
      integer begin,size,pbegin
c
c
      save begin
            if (var) then
                        begin=1
                        var=.false.
            else
            if (begin.ge.(n-size)) then
                        go=.false.
                else
                        begin=begin+size
                endif
            endif
      pbegin=begin
      return
      end
c
c
c
      subroutine dmatvet(a,x,y)
      real a(2,2)
      real x(2),y(2)
      integer i,k
c
      do 20 i=1,2
         y(i)=0.0
         do 10 k=1,2
            y(i)=y(i)+a(i,k)*x(k)
10       continue
20    continue
c
      return
      end
c
c
c
      subroutine matvet(a,v,c,nn)
      integer ndim
      parameter(ndim=1024)
c
      real a(ndim,nn)
      real v(nn),c(nn)
      real sum
      integer ii,jj
c
      do 10 ii=1,nn
         sum=0.0
         do 9 jj=1,nn
            sum=sum+a(ii,jj)*v(jj)
9        continue
      c(ii)=sum
10    continue
c
      return
      end
c
```