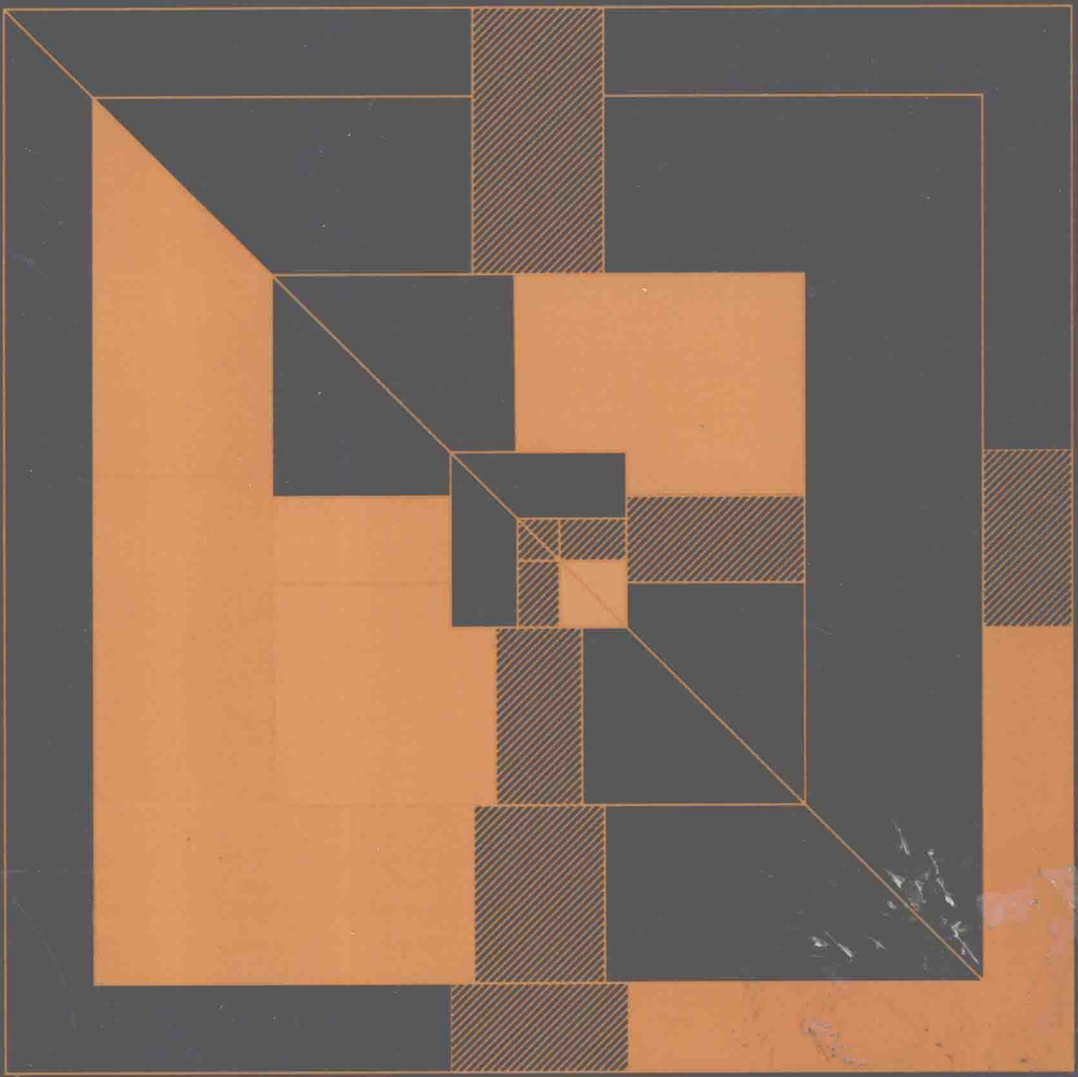
Solving Linear Systems on Vec
and Shared Memory Compute

Jack J. Dongarra, Iain S. Duff,
Danny C. Sorensen, and Henk A. van der Vorst



siam

Solving Linear Systems on Vector and Shared Memory Computers

Jack J. Dongarra
University of Tennessee and
Oak Ridge National Laboratory

Iain S. Duff
Rutherford Appleton Laboratory,
CERFACS, and
University of Strathclyde

Danny C. Sorensen
Rice University

Henk A. van der Vorst
Utrecht University

siam® *Philadelphia*

Society for Industrial and Applied Mathematics

Library of Congress Cataloging-in-Publication Data

Solving linear systems on vector and shared memory computers / Jack J.
Dongarra ... [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 0-89871-270-X
1. Algebras, Linear–Data processing. 2. Vector processing
(Computer science) 3. Parallel processing (Electronic computers)
I. Dongarra, J. J.
QA184.S65 1991
512'.5–dc20 90-24045

Preface

The purpose of this book is to unify and document in one place many of the techniques and much of the current understanding about solving systems of linear equations on vector and shared-memory parallel computers. This book is not a textbook, but it is meant to provide a fast entrance to the world of vector and parallel processing for these linear algebra applications. We intend this book to be used by three groups of readers: graduate students, researchers working in computational science, and numerical analysts. As such, we hope this book can serve both as a reference and as a supplement to a teaching text on aspects of scientific computation.

The book is divided into four sections: (1) introduction to terms and concepts, including an overview of the state of the art for high-performance computers and a discussion of performance evaluation (Chapters 1-4); (2) direct solution of dense matrix problems (Chapter 5); (3) direct solution of sparse matrix problems (Chapter 6); and (4) iterative solution of sparse matrix problems (Chapter 7). Any book that attempts to cover these topics must necessarily be somewhat out of date before it appears, because the area is in a state of flux. We have purposely avoided highly detailed descriptions of popular machines and have tried instead to focus on concepts as much as possible; nevertheless, to make the description more concrete, we do point to specific computers.

Rather than include a floppy disk containing the software described in the book, we have included a pointer to *netlib*. The problem with floppies in books is that they are never around when one needs them, and the software may undergo changes to correct problems or incorporate new ideas. The software included in *netlib* is in the public domain and can be used freely. With *netlib* we hope to have up-to-date software available at all times. A directory in *netlib* called *ddsv* contains the software, and Appendix A of this book discusses what is available and how to make a request from *netlib*.

This book only touches on topics relating to massively parallel SIMD computers and distributed-memory machines, partly because our experience lies in shared-memory architectures and partly because the areas of massively parallel and distributed-memory computing are still rapidly changing. We express appreciation to all those who helped in the preparation of this work, in particular to Gail

Pieper for her tireless efforts in proofreading drafts and improving the quality of the presentation; Ed Anderson, Mary Drake, Jeremy Du Croz, Peter Mayes, Esmond Ng, Al Geist, Giuseppe Radicati, and Charlie Van Loan for their help in proofreading and their many suggestions to improve the readability; and Reed Wade for his assistance in preparing the figures.

Contents

Introduction

The recent availability of advanced-architecture computers has had a very significant impact on all spheres of scientific computation including algorithm research and software development in numerical linear algebra. This book discusses some of the major elements of these new computers and indicates some recent developments in sparse and full linear algebra that are designed to exploit these elements.

The two main novel aspects of these advanced computers are the use of vectorization and parallelism, although how these are accommodated varies greatly between architectures. The first commercially available vector machine to have a significant impact on scientific computing was the CRAY-1, the first machine being delivered to Los Alamos in 1976. Thus, the use of vectorization is by now quite mature, and a good understanding of this architectural feature and general guidelines for its exploitation are now well established. However, the first commercially viable parallel machine was the Alliant in 1985, and more massively parallel machines did not appear on the marketplace until 1988. Thus, there remains a relative lack of definition and maturity in this area, although some guidelines on the exploitation of parallelism are beginning to emerge.

We are algebraists rather than computer scientists; as such, one of our intentions in writing this book is to provide the computing infrastructure and necessary definitions to guide the computational scientist and, at the very least, to equip him or her with enough understanding to be able to read computer documentation and appreciate the influence of some of the major aspects of novel computer design. The majority of this basic material is covered in Chapter 1, although we address further aspects related to implementation and performance in Chapters 3 and 4. In such a volatile marketplace it is not sensible to concentrate too heavily on any specific architecture or any particular manufacturer, but we feel it is useful to illustrate our general remarks by reference to some currently existing machines. This we do in Chapter 2 and Appendix C, as well as in Chapter 4 where we present some performance profiles for current machines.

It would be neither practical nor sensible to cover all aspects of parallelism in a book of this size. Instead, we have concentrated on the more well-established area of shared-memory architectures, giving only outline information on distributed-memory and massively parallel architectures.

Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. We thus concentrate on this area in this book, examining algorithms and software for dense coefficient matrices in Chapter 5 and for sparse systems in Chapters 6 and 7, where we discuss direct and iterative methods of solution, respectively. Although we have concentrated on this aspect of linear algebra, many of our observations and techniques extend to other areas—for example, the eigenproblem or the solution of least-squares problems, of which brief mention is made in Section 5.5.

Within scientific computation, parallelism can be exploited at several levels. At the highest level a problem may be subdivided even before its discretization into a linear (or nonlinear) system. This technique, typified by domain decomposition, usually results in large parallel tasks ideal for mapping onto a distributed-memory architecture. In keeping with our decision to minimize machine description, we refer only briefly to this form of algorithmic parallelism in the following, concentrating instead on the solution of the discretized subproblems. Even here, more than one level of parallelism can exist—for example, if the discretized problem is sparse. We discuss sparsity exploitation in Chapters 6 and 7.

Our main algorithmic paradigm for exploiting both vectorization and parallelism in the sparse and the full case is the use of block algorithms, particularly in conjunction with highly tuned kernels for effecting matrix-vector and matrix-matrix operations. We discuss the design of these building blocks in Section 5.1 and their use in the solution of dense equations in the rest of Chapter 5. We discuss their use in the solution of sparse systems in Chapter 6, particularly Sections 6.4 and 6.5.

As we said in the Preface, this book is intended to serve as a reference and as a supplementary teaching text for graduate students, researchers working in computational science, and numerical analysts. At the very least, the book should provide background, definitions, and basic techniques so that researchers can understand and exploit the new generation of computers with greater facility and efficiency.

Chapter 1

Vector and Parallel Processing

In this chapter we review some of the basic features of traditional and advanced computers. The review is not intended to be a complete discussion of the architecture of any particular machine or a detailed analysis of computer architectures. Rather, our focus is on certain features that are especially relevant to the implementation of linear algebra algorithms.

1.1 Traditional Computers and Their Limitations

The traditional, or conventional, approach to computer design involves a single instruction stream. Instructions are processed sequentially and result in the movement of data from memory to functional unit and back to memory. Specifically,

- a scalar instruction is fetched and decoded,
- addresses of the data operands to be used are calculated,
- operands are fetched from memory,
- the calculation is performed in the functional unit, and
- the resultant operand is written back to memory.

As demands for faster performance increased, modifications were made to improve the design of computers. It became evident, however, that a number of factors were limiting potential speed: the switching speed of the devices (the time taken for an electronic circuit to react to a signal), packaging and interconnection delays, and compromises in the design to account for realistic tolerances of parameters in the timing of individual components. Even if a dramatic improvement could be made in any of these areas, one factor still limits performance: *the speed of light.* Today's supercomputers have a cycle time on the order of nanoseconds. The CRAY-2, for example, has a cycle time of 4.1 nsec, and Cray Computer Company has announced machines with an expected cycle time of 1 nsec. One nanosecond translates into the time it takes light to move about a foot (in practice, the speed of pulses through the wiring of a computer ranges from 0.3 to 0.9 foot per nanosecond). Faced by this fundamental limitation, computer designers have begun moving in the direction of parallelism.

1.2 Parallelism within a Single Processor

Parallelism is not a new concept. In fact, Hockney and Jesshope point out that Babbage's analytical engine in the 1840s had aspects of parallel processing [94].

1.2.1 Multiple Functional Units

Early computers had three basic components: the main memory, the central processing unit (CPU), and the I/O subsystem. The CPU consisted of a set of registers, the program counter, and one arithmetic and logical unit (ALU), where the operations were performed one function at a time. One of the first approaches to exploiting parallelism involved splitting up the functions of the ALU—for example, into a floating-point addition unit and a floating-point multiplication unit—and having the units operate in parallel.

In order to take advantage of the multiple functional units, the software (e.g., the compiler) had to be able to schedule operations across the multiple functional units to keep the hardware busy. Also, the overhead in starting operations on the multiple units had to be small relative to the time spent performing the operations. Once computer designers had added multiple functional units, they turned to investigating better ways to interconnect the functional units, in order to simplify the flow of data and to speed up the processing of data.

1.2.2 Pipelining

Pipelining is the name given to the segmentation of a functional unit into different parts, each of which is responsible for partial decoding/interpretation and execution of an operation.

The concept of pipelining is similar to that of an assembly line process in an industrial plant. Pipelining is achieved by dividing a task into a sequence of smaller tasks, each of which is executed on a piece of hardware that operates concurrently with the other stages of the pipeline (see Figures

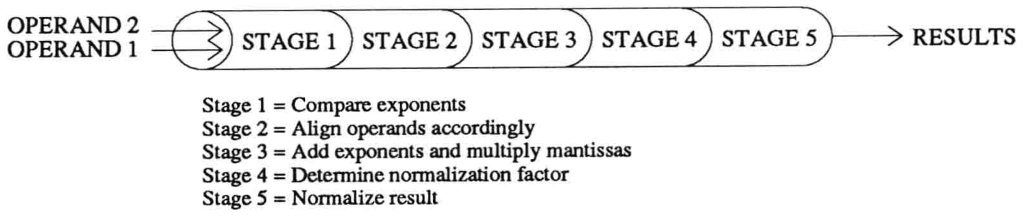

OPERAND 2
OPERAND 1
STAGE 1
STAGE 2
STAGE 3
STAGE 4
STAGE 5
RESULTS
Stage 1 = Compare exponents
Stage 2 = Align operands accordingly
Stage 3 = Add exponents and multiply mantissas
Stage 4 = Determine normalization factor
Stage 5 = Normalize result

Figure 1.1: **A simplistic pipeline for floating-point multiplication**

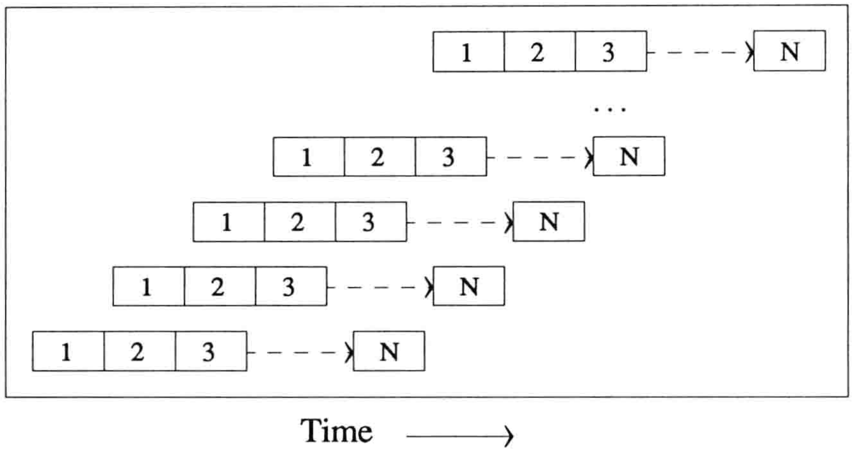


Figure 1.2: **Pipelined execution of an *N*-step process**

1.1-1.3). Successive tasks are streamed into the pipe and get executed in an overlapped fashion with the other subtasks. Each of the steps is performed during a clock period of the machine. That is, each suboperation is started at the beginning of the cycle and completed at the end of the cycle. The technique is as old as computers, with each generation using ever more sophisticated variations. An excellent survey of pipelining techniques and their history can be found in Kogge [111].

Pipelining was used by a number of machines in the 1960s, including the CDC 6600, the CDC 7600, and the IBM System 360/195. Later, Control Data Corporation introduced the STAR 100 (subsequently the CYBER 200 series), which also used pipelining to gain a speedup in instruction execution. In the execution of instructions on machines today, many of the operations are

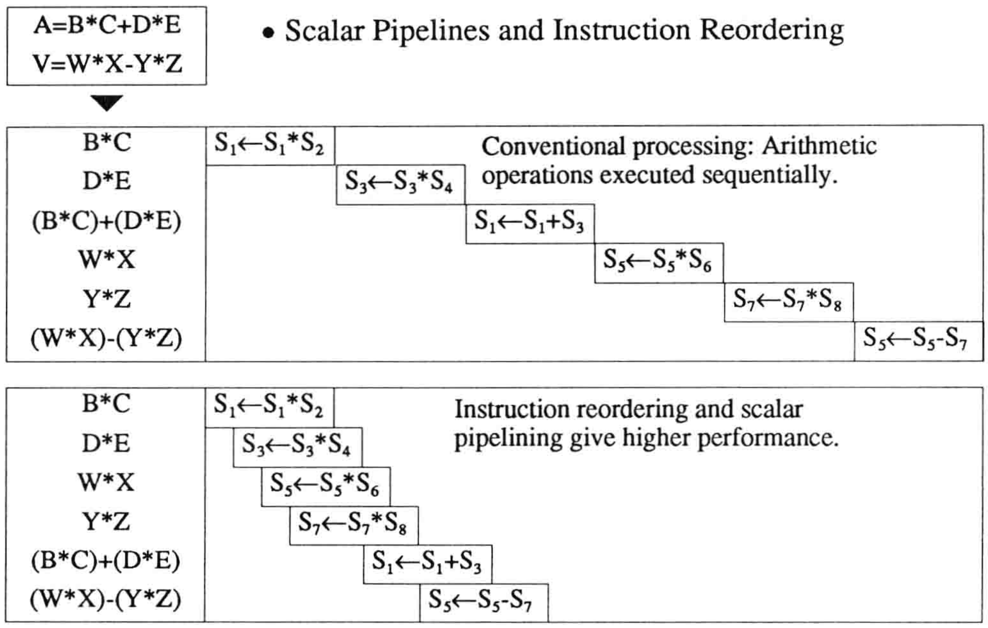

A=B*C+D*E
V=W*X-Y*Z
• Scalar Pipelines and Instruction Reordering
B*C
D*E
(B*C)+(D*E)
W*X
Y*Z
(W*X)-(Y*Z)
S1←S1*S2
S3←S3*S4
S1←S1+S3
S5←S5*S6
S7←S7*S8
S5←S5-S7
Conventional processing: Arithmetic operations executed sequentially.
B*C
D*E
W*X
Y*Z
(B*C)+(D*E)
(W*X)-(Y*Z)
S1←S1*S2
S3←S3*S4
S5←S5*S6
S7←S7*S8
S1←S1+S3
S5←S5-S7
Instruction reordering and scalar pipelining give higher performance.

Figure 1.3: **Scalar pipelines**

pipelined—including instruction fetch, decode, operand fetch, execution, and store.

The execution of a pipelined instruction incurs an overhead for filling the pipeline. Once the pipeline is filled, a result appears every clock cycle. The overhead or *startup* for such an operation depends on the number of stages or segments in the pipeline.

1.2.3 Overlapping

Some architectures allow for the *overlap* of operations if the two operations can be executed by independent functional units. *Overlap* is similar but not identical to pipelining. Both employ the idea of subfunction partitioning, but in slightly different contexts. Pipelining occurs when *all* of the following are true:

- Each evaluation of the basic function (for example, floating-point addition and multiplication) is independent of the previous one.