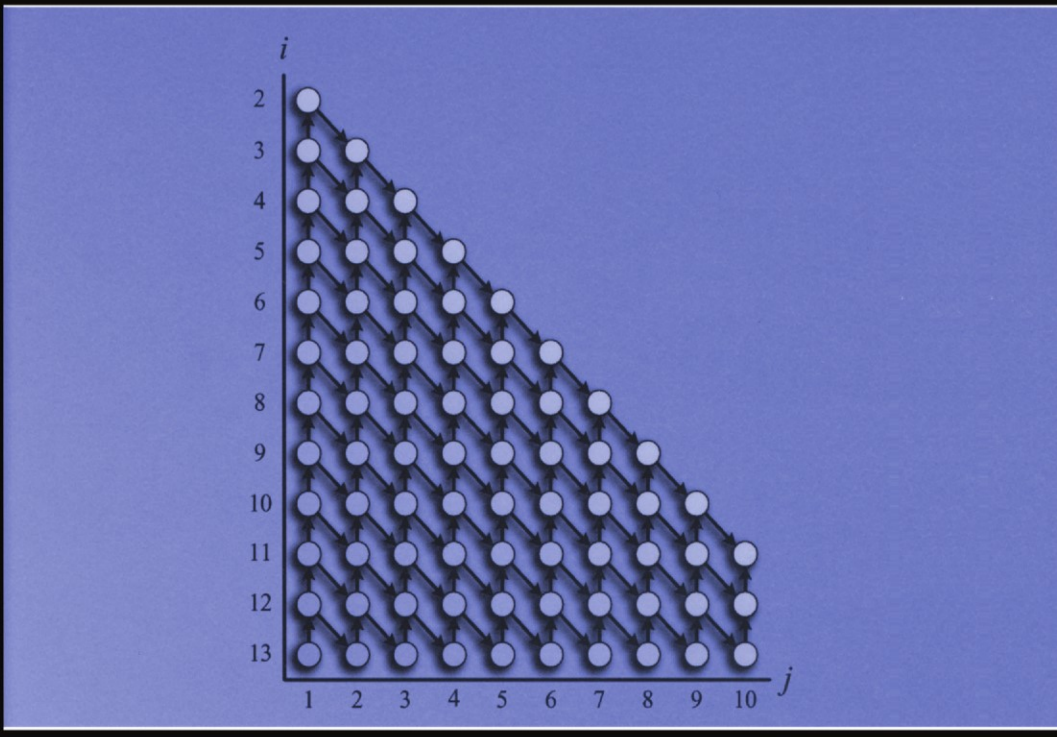
SERIES IN COMPUTER SCIENCE

A Parallel Algorithm Synthesis Procedure for High-Performance Computer Architectures



Ian N. Dunn and Gerard G. L. Meyer

A Parallel Algorithm Synthesis Procedure for High-Performance Computer Architectures

SERIES IN COMPUTER SCIENCE

Series Editor: Rami G. Melhem

University of Pittsburgh
Pittsburgh, Pennsylvania

ENGINEERING ELECTRONIC NEGOTIATIONS
A Guide to Electronic Negotiation Technologies for the Design and Implementation of Next-Generation Electronic Markets—Future Silkroads of eCommerce
Michael Ströbel

FUNDAMENTALS OF X PROGRAMMING
Graphical User Interfaces and Beyond
Theo Pavlidis

INTRODUCTION TO PARALLEL PROCESSING
Algorithms and Architectures
Behrooz Parhami

OBJECT-ORIENTED DISCRETE-EVENT SIMULATION WITH JAVA
A Practical Introduction
José M. Garrido

A PARALLEL ALGORITHM SYNTHESIS PROCEDURE FOR HIGH-PERFORMANCE COMPUTER ARCHITECTURES
Ian N. Dunn and Gerard G. L. Meyer

PERFORMANCE MODELING OF OPERATING SYSTEMS USING OBJECT-ORIENTED SIMULATION
A Practical Introduction
José M. Garrido

POWER AWARE COMPUTING
Edited by Robert Graybill and Rami Melhem

THE STRUCTURAL THEORY OF PROBABILITY
New Ideas from Computer Science on the Ancient Problem of Probability Interpretation
Paolo Rocchi

A Parallel Algorithm Synthesis Procedure for High-Performance Computer Architectures

Ian N. Dunn
Mercury Computer Systems, Inc.
Chelmsford, Massachusetts

and

Gerard G. L. Meyer
Johns Hopkins University
Baltimore, Maryland

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-4658-6 ISBN 978-1-4419-8650-4 (eBook)
DOI 10.1007/978-1-4419-8650-4

Originally published by Kluwer Academic / Plenum Publishers, New York in 2003
Softcover reprint of the hardcover 1st edition 2003

http://www.wkap.nl

10 9 8 7 6 5 4 3 2 1

A C.I.P. record for this book is available from the Library of Congress

Contents

List of Figures

List of Tables

Acknowledgments

This book grew out of research at the Parallel Computing and Imaging Laboratory. Much of the material in this book builds upon research conducted by students and members of the laboratory staff including Yosef Brandriss, Dr. James Carrig, Dr. Mike Pascale, Thomas Steck, and Dakasorn Ubol. Dr. Pascale introduced QR factorization and the critical role it plays in adaptive beamforming (Meyer and Pascale, 1995). Dr. Carrig developed a strategy for designing register and cache efficient linear algebra algorithms. His methodology led to the development of two new QR factorization algorithms (Carrig and Meyer, 1997; Carrig and Meyer, 1999). We express our sincere appreciation to Yosef Brandriss, Thomas Steck, and Dakasorn Ubol for their invaluable assistance in the development of this material as well.

This book references the research of a number of luminaries in the field of linear algebar including Christian Bischof, James Demmel, Jack Dongarra, Kyle Gallivan, Gene Golub, Richard Hanson, William Kahan, Charles Lawson, Robert Schreiber, and Charles Van Loan. Their work created a solid mathematical foundation on which to develop the material in this book.

Finally, we thank our families for their constant encouragement and patience.

Chapter 1

INTRODUCTION

Parallel computing is the only viable, cost-effective approach to meeting the timing constraints of many high performance signal processing applications. The computational and/or I/O requirements of a single application can overwhelm the capabilities of a sequential computer. Applications in array signal processing and image processing perform complex sequences of matrix computations, have streaming data requirements in excess of one Gbits/s, and must execute in 10 ms or less. The operating constraints of these applications mandate the use of parallel computing.

Despite widespread predictions to the contrary, the pace of progress in commercial, off-the-shelf processor technology has not slowed down. This progress keeps the number of applications in the realm of parallel computing relatively small and relegates parallel computing to the fringes of mainstream computing. As a consequence, parallel algorithm designers lack standard software tools for designing, debugging, and benchmarking parallel algorithms. They must rely heavily upon intuition, a deep knowledge of the underlying parallel architecture, vendor-specific optimizing compiler technologies, and vendor-tuned kernel libraries for basic mathematical operations. In addition, the limited success and short life of many parallel computers forces algorithm designers to rapidly master new architectures and development environments.

While the vagaries of the commercial marketplace for parallel computers make for a challenging development environment, the real problem for algorithm designers remains devising a program to distribute an algorithm across multiple processors and share intermediate results to satisfy dependencies among computations. Various programming models have been proposed for distributing algorithms and managing shared data across multiple processors. Within the confines of these programming models, systematic techniques for minimizing communication among processors, minimizing the number of

times all processors must stop to synchronize, or maximizing the parallelism in an algorithm abound. Unfortunately, the successive application of a handful of these techniques can produce parallel algorithms that run even slower than their sequential counterparts.

To navigate this challenging environment, algorithm designers need a road map, a detailed procedure that designers can use to efficiently develop high performance, portable parallel algorithms. The focus of this book is to draw such a road map. The Parallel Algorithm Synthesis Procedure can be used to design reusable building blocks of adaptable, scalable software modules from which high performance signal processing applications can be constructed. The hallmark of the procedure is a semi-systematic process for introducing parameters to control the partitioning and scheduling of computation and communication. This facilitates the tailoring of software modules to exploit different configurations of multiple processors, multiple floating-point units, and hierarchical memories. To showcase the efficacy of this new procedure, Chapters 5, 6, and 7 describe in detail three new adjustable algorithms for matrix factorization.

1. Notation and Conventions

Throughout this book, algorithms are defined using a stylized variant of the MATLAB programming language similar to the one used by Golub and Van Loan, 1989. The level of detail is designed to make translation to any sequential higher-level language such as C, C++, or FORTRAN systematic.

All matrices are described using capital letters, and vectors are described using lower case letters. Subscripts are used to describe elements of vectors or matrices, and colon notation used in the subscript describes a range of elements. For instance, $A_{11:14,3}$ is used to describe a small column vector with elements $A_{11,3}$, $A_{12,3}$, $A_{13,3}$, and $A_{14,3}$. Very little distinction is made between mathematical notation and programming notation. The only important difference is the addition of a superscript index to distinguish between different versions of the same scalar, vector, or matrix. For instance,

$$A^i = (I + YS^TY^T)A^{i-1}$$

expresses a computation whereby the contents of the matrix A are replaced by the contents of the product

$$(I + YS^TY^T)A.$$

To describe numerical procedures, the concept of an *algorithm* and a *procedure* is introduced. An algorithm has a clearly defined input/output relationship. A procedure, on the other hand, is a recipe for accomplishing a task. While the procedure may also have some input/output relationships,

its primary role is to accomplish a task that cannot be strictly described by its terminal output condition. To make this distinction clear, the following examples of an algorithm and a procedure are provided.

Algorithm: Matrix-Vector Multiplication

Input(A, x)

$[m, n] =$ dimensions(A)

For $i = 1$ to m

$y_i = A_{i,1:n}x$

End For

Output(y)

Procedure: Message Passing

Step 1: If p is ODD and $p = 0$ then go to Step

Step 2: If p is EVEN and $p = P$ then go to Step

Step 3: If p is ODD then send A to $p - 1$; else receive A from $p + 1$

Step 4: Stop.

For parallel algorithms, we define the following extensions:

DO FOR-LOOP IN PARALLEL – execute elements of the ensuing "for-loop" in parallel

SEND – send a message to another processor

RECV – receive a message from another processor.

2. Chapter Organization

The remainder of this book is organized as follows. Chapter 2 introduces some key aspects of parallel computing including architecture, parallel programming environments, and performance metrics. Chapter 3 formalizes a design methodology for parallel algorithms: the Parallel Algorithm Synthesis Procedure. The synthesis procedure is applied to fast Givens QR factorization, Compact WY QR factorization, and matrix bidiagonalization in Chapters 5, 6, and 7, respectively. Chapter 4 contains a review of standard Givens-based and Householder-based factorization algorithms. Chapter 8 presents final conclusions.

Chapter 2

PARALLEL COMPUTING

Despite five decades of research, parallel computing remains an exotic, frontier technology on the fringes of mainstream computing. Its much-heralded triumph over sequential computing has yet to materialize. This is in spite of the fact that the processing needs of many applications continue to eclipse the capabilities of sequential computing.

The culprit is largely the software development environment. Fundamental shortcomings in the development environment of many parallel computer architectures thwart the adoption of parallel computing. Foremost, parallel computing has no unifying model to accurately predict the execution time of algorithms on parallel architectures. Cost and scarce programming resources prohibit deploying multiple algorithms and partitioning strategies in attempt to find the fastest solution. As a consequence, algorithm design is largely an intuitive art form dominated by practitioners who specialize in a particular computer architecture. In addition, parallel computer architectures rarely last more than a couple of years. Porting an algorithm to a new architecture often requires extensive retuning, or in some cases a completely new implementation, to accommodate dissimilar programming environments, interconnection networks, and processor technologies. The availability of a unifying model to accurately predict execution time would address both these shortcomings by promoting the development of standard software tools for designing, debugging, and benchmarking parallel algorithms. This would greatly reduce the effort involved in porting algorithms to new architectures. In the absence of such a model, parallel algorithm designers must rely on intuition and hands-on experience to manage a complex and challenging design environment.

To put the parallel algorithm design problem into perspective, this chapter reviews some key aspects of parallel computer architecture, the two predominant parallel programming models (*shared memory* and *message passing*),

and various performance metrics that algorithm designers must employ in the design and implementation of an algorithm.

1. Architectures

If a single processor can solve a problem in 10 seconds, can 10 processors working in harmony solve the problem in one second? Since its inception, the field of parallel computing has been struggling with this question. The struggle has taken many architectural forms in the last four decades. Some noteworthy architectures include:

1964 - Control Data Corporation CDC-6600

1966 - IBM 360/91

1969 - Control Data Corporation CDC-7600

1970 - MIT and DEC produce PDP-6/KA10

1971 - Control Data Corporation Cyberplus

1972 - Goodyear STARAN, Burroughs PEPE

1974 - IBM 3838

1976 - Cray Research Cray-1, Control Data Corporation Flexible Processor, Floating Point Systems AP-120B

1981 - BBN Butterfly, DEC VAX-11, Control Data Corporation Cyber 205, Floating Point Systems FPS-124

1982 - Cray Research Cray X/MP, Denelcor HEP, Control Data Corporation Advanced Flexible Processor

1983 - Fujitsu VP-200, Goodyear Aerospace MPP

1985 - IBM 3090, Intel iPSC/1, NEC SX-2, NCube NCube/10, Floating Point Systems FPS-264, Convex C1, Cray Research Cray-2

1986 - Thinking Machines Corporation CM-1

1988 - Silicon Graphics POWER Series, Cray Research Y/MP, Intel iPSC/2, Hitachi S-820, FPS 500 (1988)

1989 - Fujitsu VP2000, NCube NCube/2

1990 - MasPar MP-1, NEC SX-3, Fujitsu VP2600, Intel iPSC/860, Cray Research C90

1991 - Kendall Square Research KSR-1, Think Machines Corporation CM-200, Intel iWarp, Intel Paragon

1992 - MasPar MP-2, Thinking Machines Corporation CM-5

1993 - IBM PowerParallel, NEC Cenju-3.

The vast majority of the companies in this abridged timeline, which was culled from Gregory Wilson's "The History of Parallel Computing" (Wilson, 1993), have since become extinct. In the seven years since this timeline was compiled, the field of parallel computing has witnessed one of the most important transitions in its history. It has progressed from being a field dominated by supercomputing vendors to one where PC and workstation vendors such as HP, IBM, Intel, SGI, and Sun Microsystems now participate on a level playing field with the few remaining supercomputing vendors. This transition ushered in the end of single-chassis parallel computing and marked the advent of multi-chassis parallel computing or distributed/cluster computing built around smaller, single-chassis parallel computers.

The driving force behind this transition has been a growing, robust marketplace for small-scale multiprocessors where several processors share a single physical memory. Vendors employ *superscalar* processor technology and large *caches* to reduce the demands on the shared memory bus while still achieving high levels of performance. Superscalar processors are capable of performing multiple scalar operations per clock cycle and are, in effect, mini parallel computers. The caches complement the superscalar cores by providing fast on-chip buffers to store commonly used data. The resulting shared memory multiprocessor is an extremely versatile and cost effective platform for a variety of applications including computation-intensive database and server operations.

To leverage this commercial, off-the-shelf parallel computing technology for more demanding applications in science and engineering, multiple shared memory multiprocessors are linked together through various interconnection networks to build systems with hundreds, and in some cases, thousands of processors. These systems are generally referred to as *distributed shared memory* (DSM) multiprocessors (Judge et al., 1999; Bell and van Ingen, 1999; Protic et al., 1996), and they dominate the marketplace for large- and small-scale parallel computers.

Only a handful of parallel computer vendors continue to conduct research in the areas of processor architecture and interconnection networks. In fact in the area of processor architecture, the number of vendors is likely to narrow even further when parallel computers based on the second generation IA-64 architecture become available. Interconnection network technology stills vary widely by vendor. The advent of standard 2.5 Gbaud serial transmission technologies will narrow the spectrum of available interconnect technologies

with Gigabit and 10 Gigabit Ethernet playing the dominant role. Vendors have settled on a few popular network topologies including the fat tree (more generally, multistage interconnection networks), the 1-D torus (ring), the 2-D grid, the 2-D mesh, and the hypercube. For example Figures 2.1 and 2.2 show the fat tree and the hypercube. The shaded nodes are in general adaptive and can choose different routes to avoid blocking or collisions. The complex and proprietary routing algorithms at the heart of these adaptive nodes make developing models for these devices almost impossible. Network models based on traditional topological structures are no longer accurate predictors of performance.

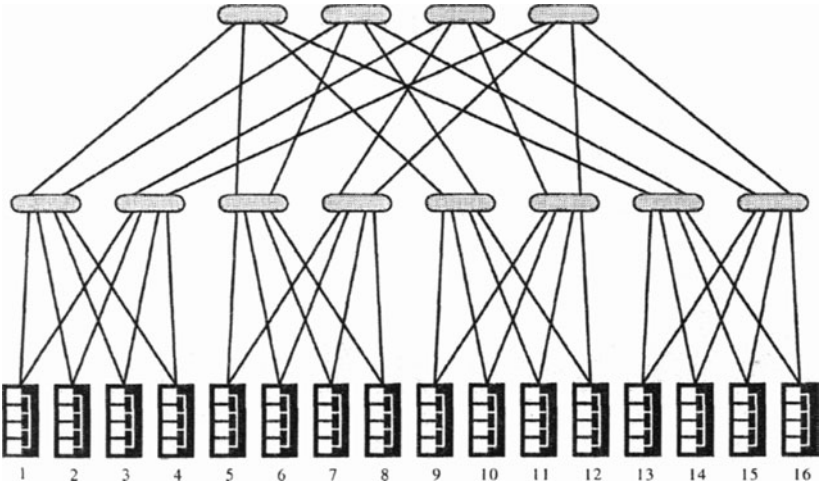


Figure 2.1. A fat tree interconnection network for 16 shared memory multiprocessors. Each shared memory multiprocessor is comprised of four processors.

In this book, numerical experiments are conducted on four distributed shared memory architectures: a 64-processor HP SPP-2000, a 32-processor HP V2500, a 32-processor IBM SP3, and a 128-processor SGI Origin 2000. The HP SPP-2000 is comprised of four shared memory multiprocessors or four hypernodes. Each hypernode contains 16 180 MHz PA-8000 superscalar processors. The PA-8000 is capable of executing two multiply-accumulate operations every clock cycle. Peak performance is 720 Mflops. Only a single cache level is provided, and it can hold up to one MByte of data. Within a hypernode, the interconnection fabric is built around a crossbar network that links 16 processors to eight memory access controllers. Up to four hypernodes are connected in multiple rings. The HP V2500 has a very similar architecture to the SPP-2000. The main difference is that at the heart of the

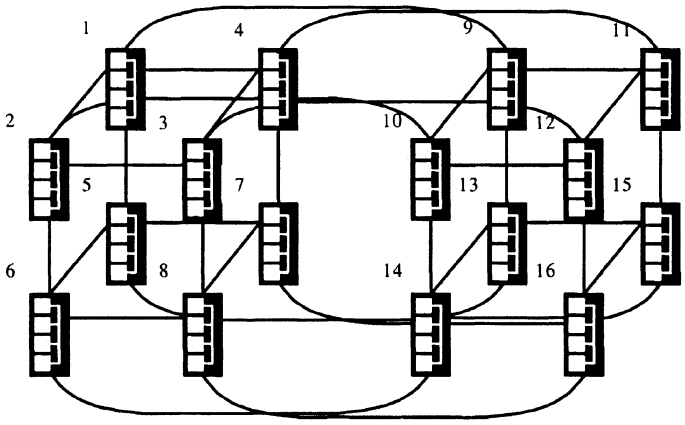


Figure 2.2. A hypercube interconnection network for 16 shared memory multiprocessors. Each shared memory multiprocessor is comprised of four processors.

V2500 are 32 440 MHz PA-8500 superscalar processors. These processors have a peak performance of 1760 Mflops. The IBM SP3 is comprised of four shared memory multiprocessors or nodes. Each node contains eight 222 MHz Power3 processors. The Power3 has a peak performance of 888 Mflops. Multiple nodes are connected using a multistage interconnection network similar in structure to a fat tree. The level-1 cache can store 64 KBytes of data, and the level-2 has a four MByte unified instruction/data cache. The SGI Origin 2000 is comprised of 128 300 MHz R12000 superscalar processors. Each processor is capable of executing a multiply-accumulate operation every clock cycle and can therefore deliver as much as 600 MFLOPS of performance. The level-1 cache can store 32KBytes of data, and the level-2 has an eight MByte unified instruction/data cache. Two processors and a hub controller comprise a node, and multiple nodes are connected in a hypercube. The hub controller manages data traffic between nodes.

2. Programming Models

In large-scale systems, memory is almost always physically distributed. The two predominant programming models for managing distributed, stored program data are the *shared memory model* and the *distributed memory model*. In the shared memory model, all stored data is globally accessible to all

processors. The partitioning and scheduling of communication are transparent to the algorithm designer. These functions are managed by the hardware and system software as a function of the real-time data requirements of the executing algorithm. In the distributed memory model, distributed data is only accessible to the local processor. The algorithm designer is responsible for explicitly transferring stored data among processors.

While the shared memory model provides for a much simpler programming environment, algorithm designers have no control over how communications are partitioned or when they are sent over the network. The executing program triggers transfers implicitly via accesses to the memory hierarchy. The partitioning of communication is a function of how the data is physically stored in memory. As for distributed memory, the algorithm designer has some degree of control over how and when messages are sent over the interconnection network. The amount of control depends on the functionality and implementation of available message passing libraries.

To enhance portability in the two programming environments, two standard shared memory and message passing libraries have been proposed: OpenMP (Dagnum and Menon, 1998; OpenMP Architecture Review Board, 1999; Throop, 1999) and the Message Passing Interface (MPI, Message Passing Interface Forum, 1997). Both libraries are widely accepted and supported by the majority of the vendors including IBM and SGI. In addition, there has been some discussion of developing a unified standard library specification (Bova et al., 1999). This would allow algorithm designers to blend (*hybrid shared memory/message passing*) the two programming environments in a single executable without sacrificing portability. Currently, blending is possible on some architectures, but the library settings and compiler flags necessary to accomplish this are vendor specific. As a consequence, few researchers have examined the performance benefits of this feature. To shed some light on the potential benefits of such a feature, the case study in Chapter 5 investigates the performance characteristics of a hybrid programming environment.

3. Performance Metrics

One consequence of not having a unifying model to accurately predict the execution time of algorithms on parallel architectures is that algorithm designers must rely on a patchwork of performance metrics to guide the design process, to evaluate various algorithm-architecture pairs, and to gain some insight as to how the performance of an algorithm-architecture pair may be improved. Algorithm designers who rely on a single metric run the risk of introducing unintended biases into their design process. This section explores the advantages and disadvantages of some of the most common performance metrics in parallel computing: latency, throughput, speedup, and efficiency.

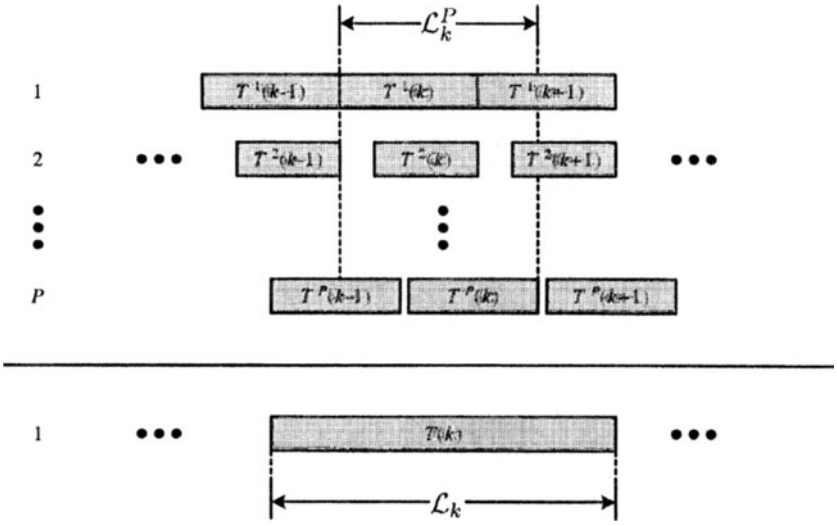


Figure 2.3. Parallel latency $\mathcal{L}^P$ and sequential latency $\mathcal{L}$ for the k^{th} execution of an algorithm.

Of the performance metrics discussed in this section, none should be used independently of latency. It provides the most unbiased measurement for comparing algorithm-architecture pairs. Throughput, speedup, and efficiency matter insofar as they favor algorithm-architecture pairs with minimum latency or provide some insight as to how the performance of an algorithm-architecture pair may be improved.

Latency is defined as the elapsed time from initial data input to final data output and is denoted by the symbol $\mathcal{L}$. Parallel and sequential latency for the k^{th} execution of an algorithm is shown in Figure 2.3. Unfortunately for parallel algorithms, latency measurements reveal nothing about the initial distribution of the input data or final distribution of the output data. As a consequence, algorithms can hide some of their communication costs by predistributing the data and ignoring any costs associated with the final distribution pattern of the output data. To remedy this situation, the real cost of redistributing the input and output data from a fixed distribution scheme should be included in the latency measurements. If possible, the scheme should be representative of the data distribution requirements of a target application.

The *throughput* of a parallel algorithm $\mathcal{T}$ is the number of operations computed per second, or the ratio of the number of arithmetic operations $\mathcal{A}$

of an algorithm to the its *execution time* $\mathcal{T} = \max(\{\mathcal{T}_k\})]$. Throughput for the k^{th} execution step is shown in Figure 2.4.

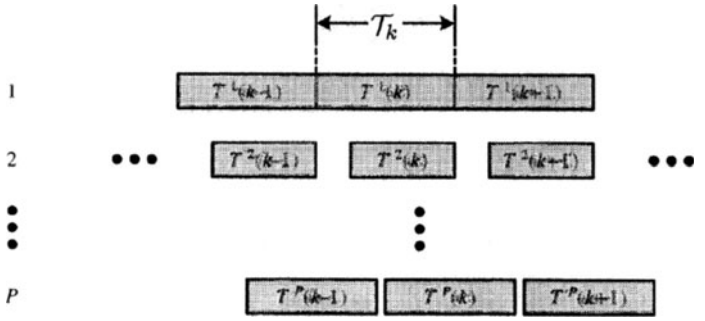


Figure 2.4. Throughput time $\mathcal{T}_k$

In some I/O bound algorithms throughput requirements can be far more stringent than latency requirements. This is not true, however, of most embedded algorithms. Throughput can be thought of as the rate at which information must be handled, and latency can be thought of as the time allotted for making a decision based on the available information. The quality of the decision depends on the age of the information. Thus, the latency requirements of an algorithm are usually as stringent as the requirements for throughput.

Another measure of an algorithm-architecture pair is *speedup*. Speedup $\mathcal{S}^P$ is defined as the ratio of single processor latency $\mathcal{L}$ to multiprocessor latency $\mathcal{L}^P$ on P processors, that is $\mathcal{L}/\mathcal{L}^P$, generally bounded by $1 \leq \mathcal{S}^P \leq P$. A similar measure is the *efficiency* of an algorithm-architecture pair, or normalized speedup. The efficiency using P processors $\mathcal{E}^P$ is the ratio of the speedup $\mathcal{S}^P$ to the number of processors P that is $\mathcal{E}^P = \mathcal{S}^P/P = \mathcal{L}/(\mathcal{L}^P P)$. Efficiency is bounded by $1/P \leq \mathcal{E}^P \leq 1$.

A fundamental shortcoming of both speedup and efficiency as performance metrics, however, is that they tend to favor inefficient compilation for uniprocessor execution. To circumvent this problem, numerous authors have defined extensions for speedup, including relative, asymptotic, relative asymptotic, and scaled speedup (Sahni and Thanvantri, 1996; Sun and Gustafson, 1991). The metric of choice in this research is latency also referred to as execution time.

Chapter 3

PARALLEL ALGORITHM SYNTHESIS PROCEDURE

The Parallel Algorithm Synthesis Procedure introduces parameters to control the partitioning and scheduling of computation and communication. The goal is to design and implement parameterized software components that can be tailored to exploit multiple scalar units within a single processor, hierarchical memories, and different configurations of multiple processors. At the heart of the synthesis procedure is a computational model that provides a qualitative framework for introducing parameters to improve reuse in the register file and memory hierarchy, balancing the load among P processors, and reducing data traffic over the processor interconnection network.

Given a numerical problem, application of the procedure begins with a high-level language description of a candidate algorithm. In general, the description contains both explicit and implicit information on the type and number of computations and the order in which these computations are to be executed. The explicit information comes from the structure of the solution algorithm as it is described in the high-level language. The implicit information is contained in language primitives for controlling program flow and in library subroutines for performing computations. The challenge to algorithm designers is to unravel both types of information while exploiting any potential freedom to reorder the component computations for efficient execution. The synthesis procedure guides the algorithm designer in tackling this challenging problem. Chapters 5, 6, and 7 employ the Parallel Algorithm Synthesis Procedure in the design of three matrix factorization algorithms. In particular, the case studies in Chapters 5 and 6 examine the problem of designing a parallel algorithm for computing the QR factorization of a real $m \times n$ matrix. The case study in Chapter 7 examines the problem of designing a parallel algorithm for computing the bidiagonal factorization of a real $m \times n$ matrix.

The case study chapters are ordered, from most comprehensive in Chapter 5 to least comprehensive in Chapter 7.

This chapter is organized as follows: before introducing the synthesis procedure, the underlying architectural model is discussed in Section 1; Section 2 introduces the Parallel Algorithm Synthesis Procedure; and in Section 3, related work is discussed, including some recent developments in optimizing compiler technology. For demonstration purposes, this chapter deploys the standard Householder QR factorization (SH) algorithm. A complete description of the algorithm can be found on page 37.

1. Architectural Model for Algorithm Synthesis

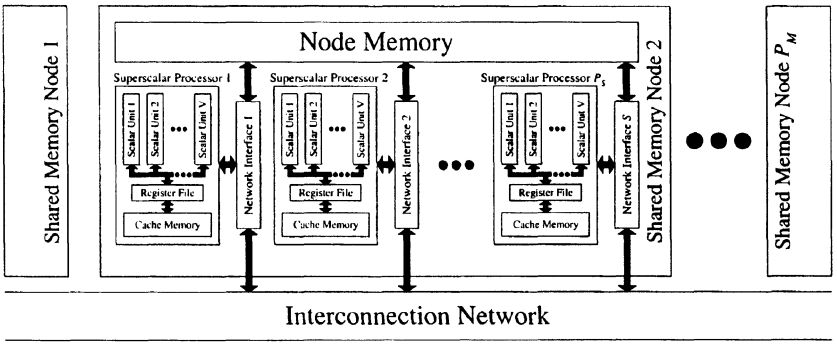


Figure 3.1. Architectural model of a distributed memory machine built from multiple shared memory multiprocessors

Figure 3.1 is a high-level depiction of the architectural model that underlies the Parallel Algorithm Synthesis Procedure. The model is an abstraction of a distributed shared memory machine built from multiple shared memory multiprocessors and is representative of an increasingly popular class of high performance computer architectures. These include the HP SPP-2000, HP V2500, IBM SP3, and SGI Origin 2000.

The architecture consists of a network of P_m shared memory multiprocessors or nodes. A node is comprised of P_s superscalar processors. The total number of processors is $P = P_m P_s$. A superscalar processor is comprised of multiple memory elements arranged in hierarchical fashion (*memory hierarchy*), one or more *scalar units* for performing computations, and a *network interface controller*. The highest level of the memory hierarchy is the *register file*. The scalar units can use any combination of registers as source and destination operands for computation. Before computation can begin in the scalar units, data must first be loaded into the register file from the level-1 cache.

The caches have a line width of one word and are fully associative - any word stored in the level-i cache or node memory can be mapped to any location in the level-$(i-1)$ cache or the level-c cache for $i = 1, 2, \ldots, c-1$. Words are expunged from the caches according to a *least-recently-used replacement policy*. The network interface controllers transfer data between processors and node memory. In the absence of resource conflicts, these transfers can be executed in parallel with computational processing.

Although memory is physically distributed across the shared memory multiprocessors, algorithm designers can choose to manage the distributed, stored data using a shared memory or message passing programming model. In the shared memory model, all stored data is globally accessible to all processors. The partitioning and scheduling of communication are transparent to the algorithm designer because they are managed by the hardware and system software as a function of the real-time data requirements of the executing algorithm. In the distributed memory model, distributed data is only accessible to network interface controllers and scalar units of the local processor. The algorithm designer is responsible for the partitioning and scheduling of both computation and communication. The executing program explicitly manages the data requirements of the algorithm. The network interface controller can manage a single send-or-receive transaction from one of the neighboring processors. The transaction can be executed simultaneously with computational processing. In the case of the shared memory model, the network interface controller determines whether a memory request is local or global, and fetches the appropriate data for the requesting cache.

For simplicity, the *interconnection network* is assumed to be a *linear array*, and each shared memory multiprocessor exchanges messages with its left and right neighboring multiprocessors only. While few machines built today employ linear array topologies, most static and dynamic interconnection networks can mimic the functionality of a linear array with minimal resource conflicts. Thus, our choice does not restrict this work to any particular class of interconnection topologies. Indeed, the machines under consideration in this book can easily accommodate a linear array topology.

2. Synthesis Procedure

The first step in the synthesis procedure is to define the *basic primitives*, which are the basic units of computation. The idea is to tailor the composition of the basic primitives to exploit characteristics of superscalar microprocessors. These characteristics are defined as the ability to perform multiple scalar or floating-point operations per clock cycle. This involves selecting the type, mixture, and number of floating-point operations that comprise a basic primitive. If possible, the mixture and type of floating-point operations should resemble the mixture and type of scalar units found in the target processor.

The total number of floating-point operations should be large enough to offset any costs associated with executing the primitives on a single processor. A candidate set of basic primitive definitions must meet the following conditions: 1) the definitions must be disjointed; 2) the union must encompass all of the algorithm's component computations; and 3) each component computation must belong to one and only one basic primitive.

For a particular solution algorithm, there may be dozens of sets of candidate basic primitive definitions. To narrow the choices, each set of candidate definitions should be profiled on the target architecture, and the amount of available parallelism for multiprocessor execution should be estimated. Unfortunately, there is generally a tradeoff between sequential and parallel performance. Superscalar processors are small parallel computers, and the severity of the tradeoff is usually related to the number of scalar units. As a consequence, careful attention must be paid to sequential performance because efficient superscalar operation can typically reduce the number of processors by as much as 50%.

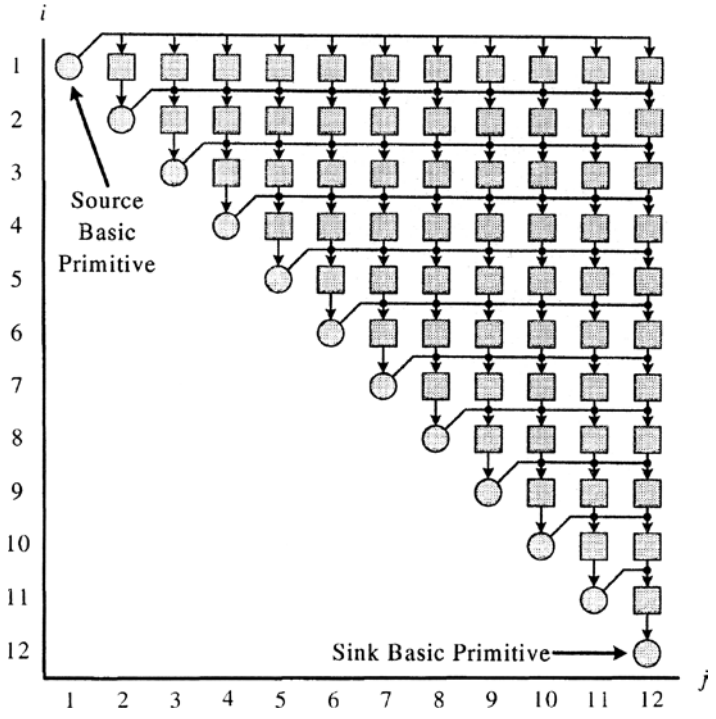


Figure 3.2. Sink and source primitives for the SH algorithm.

To gauge the amount of available parallelism given a candidate set of primitive definitions, the number of *source concurrency sets* and *sink concurrency sets*, and the cardinality of the largest source and sink concurrency sets should be determined. This is accomplished by deriving a dependency graph representation of the *ordering constraints* among the basic primitives. The ordering constraints reflect underlying, *value-based data dependencies* among basic primitives. Value-based data dependencies occur when one basic primitive touches a value and then another basic primitive subsequently touches the same value (Wolfe, 1996b). For deterministic algorithms, dependency graphs have at least one source and one sink basic primitive. A source primitive has no predecessor primitives. A sink primitive has no dependent primitives. For example, the source and sink primitives are depicted in Figure 3.2 for the dependency graph representation of the standard Householder algorithm. Source and sink concurrency sets are groups of non-overlapping, independent primitives. They are constructed iteratively using the following procedures:

Procedure: Source Concurrency Set Definition

Step 1: Let $i = 1$

Step 2: Let C_i be all source basic primitives

Step 3: Let G_{i+1} be all primitives that are dependent on basic primitives C_i

Step 4: Let C_{i+1} be the largest group of basic primitives from G_{i+1} that are independent

Step 5: If basic primitives have not been assigned to any C_k for $1 \leq k \leq i$, then let $i = i + 1$ and go to Step 3; else stop.

Procedure: Sink Concurrency Set Definition

Step 1: Let $i = 1$

Step 2: Let C_i be all sink basic primitives

Step 3: Let G_{i+1} be all primitives that are predecessor basic primitives to C_i

Step 4: Let C_{i+1} be the largest group of basic primitives from G_{i+1} that are independent

Step 5: If basic primitives have not been assigned to any C_k for $1 \leq k \leq i$, then let $i = i + 1$ and go to Step 3; else stop.

If the amount of parallelism among concurrency sets varies widely, then the size of the concurrency sets in step 4 should be modified to mitigate inequalities.

Within a source or sink concurrency set, the basic primitives are independent. Thus, they can be executed concurrently. The total number of sink or source concurrency sets provides the means for determining a rough estimate of the latency or the minimum execution time in units of the basic primitives. Sequential profiling data can be used to translate this estimate into execution time. The cardinality of the largest source and sink concurrency sets aids in estimating the number of processors needed to meet this estimate. This information, coupled with the results of the profiling, can be used to eliminate inefficient sets of basic primitive definitions.

Two basic primitives are defined in the following modified Gram-Schmidt QR factorization algorithm:

Algorithm: Modified Gram-Schmidt QR Factorization

Input(A)

$A^0 = A$

$[m, n]$ =dimensions(A^0)

For $k = 1$ to n

$A^k_{k,k} = \text{norm}(A^{k-1}_{1:m,k})$ **(Operation 1)**

$Q_{1:m,k} = A^{k-1}_{1:m,k} / A^k_{k,k}$ **(Operation 2)**

For $j = k + 1$ to n

$A^k_{k,j} = Q^T_{1:m,k} A^k_{1:m,j}$ **(Operation 3)**

$A^k_{1:m,j} = A^{k-1}_{1:m,j} - Q_{1:m,k} A^k_{k,j}$ **(Operation 4)**

End For

End For

$A = A^{k-1}$

Output(A, Q)

The "leading" basic primitive includes operations 1 and 2. The "subordinate" basic primitive includes operations 3 and 4. With the basic primitives defined, the ensuing sections help the algorithm designer introduce parameters to control the partitioning and scheduling of computation and communication.

2.1 Superscalar Parameterization

The first layer of parameterization in the synthesis procedure focuses on the problem of reducing data traffic between the memory hierarchy and the scalar units. Parameters are introduced to control the amount of *value-based*

reuse in a group of basic primitives. Value-based reuse refers to the number of times a value is used by the scalar units after it is loaded into a register and before it is either stored in memory or overwritten. This type of reuse is critical to attaining high levels of performance because the scalar units can in general consume more values per clock cycle than can be loaded into the register file per clock cycle. To improve reuse in the register file, the parameter ψ is introduced using the following:

Procedure: Superscalar parameterization

Step 1: Select a single source primitive and denote by the symbol g_1

Step 2: Assume only the values needed to execute g_1 are stored in the register file

Step 3: Set $i = 1$

Step 4: Let E_i be the set of all primitives not in the set $\{g_1, g_2, \ldots, g_i\}$ that share a path of length 1, or share no path of any length with the set of primitives $\{g_1, g_2, \ldots, g_i\}$

Step 5: Let G_i be a subset of E_i where a primitive in E_i is also in G_i if and only if all of its predecessor primitives that share a path length of 1 are in the set $\{g_1, g_2, \ldots, g_i\}$

Step 6: For each and every primitive in G_i, tabulate any additional data that will have to be loaded into the register file or stored to the level-1 cache from the register file before the primitive can be executed

Step 7: Select the primitive from G_i that requires the largest number of additional loads/stores and denote it by the symbol g_{i+1}

Step 8: Let $i = i + 1$

Step 9: Repeat steps 4-8 $\psi - 1$ times

Step 10: Set $k = i$

Step 11: Assume the register file is large enough to store only the values needed to execute $\{g_{k-\psi+1}, g_{k-\psi+2}, \ldots, g_k\}$

Step 12: Select the primitive from G_i that requires the smallest number of additional loads/stores and denote it by the symbol g_{i+1}

Step 13: Let $i = i + 1$

Step 14: Repeat steps 11-16 until the primitive g_i no longer reuses any of the values already loaded into the register file

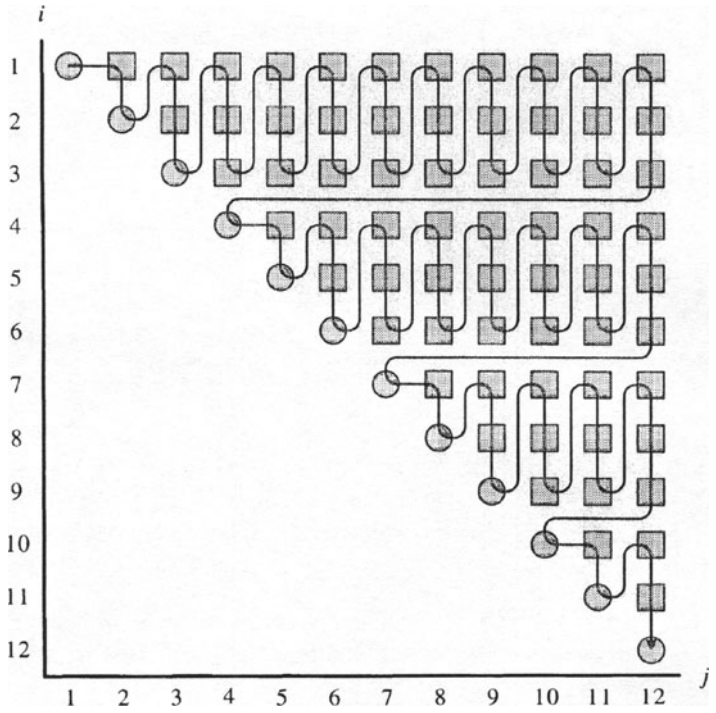


Figure 3.3. Ordering scheme 1 for the SH algorithm where $\psi = 3$, $m \geq n$, and $n = 12$.

Step 15: Return to step 6 while E_i is non-empty

Multiple primitives may satisfy the selection conditions in steps 7 and 12. As a consequence, a unique value of ψ may produce multiple, dissimilar ordering schemes. For example, the Superscalar Parameterization procedure applied to the SH algorithm produces at least two ordering schemes, and they are depicted in Figures 3.3 and 3.4. The algorithm designer should profile candidate ordering schemes on the target architecture to ascertain the merits of one scheme over another. If the basic primitives can be built from machine-optimized subroutines, then the superscalar parameterization may not be necessary. In particular, if the profiling step reveals that the basic primitives achieve the desired levels of performance, then parameterization can be skipped, and the algorithm designer should proceed directly to the memory hierarchy parameterization.

The above procedure produces the best results if the algorithm designer assumes that the register file is capable of storing enough data to execute $r\psi$

primitives where $r > 1$. While this is generally not true, the effects of this assumption on performance are negated by the next layer of parameterization.

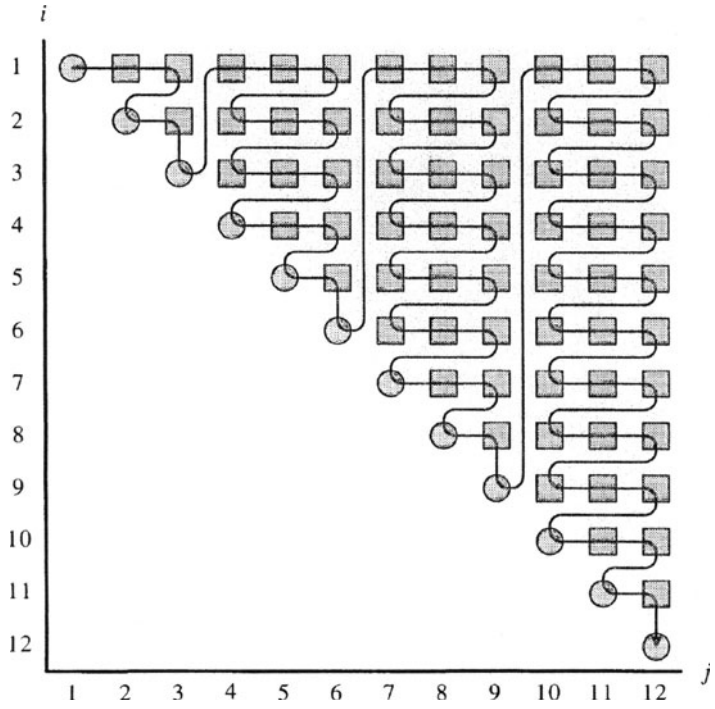


Figure 3.4. Ordering scheme 2 for the SH algorithm where $\psi = 3$, $m \geq n$, and $n = 12$.

To what extent the benefits of reuse are realized will depend on the compiler. Compilers rearrange computations to alleviate contention for registers and manage data dependencies for peak scalar unit performance. So that this rearranging can be applied efficiently, the procedure identifies small groups of computations that can be optimized for efficient execution by the compiler.

The second superscalar parameter ρ targets the compiler. The goal of the parameterization is to aggregate primitives into superscalar primitives. The aggregation of primitives should complement the ordering parameterized by ψ and provide guidance to the compiler for instruction scheduling and register allocation optimizations. The parameter ρ cuts the ordering scheme parameterized by ψ into segments. For example, the parameter ρ cuts the ordering schemes for the SH algorithm into segments as depicted in Figure 3.5.

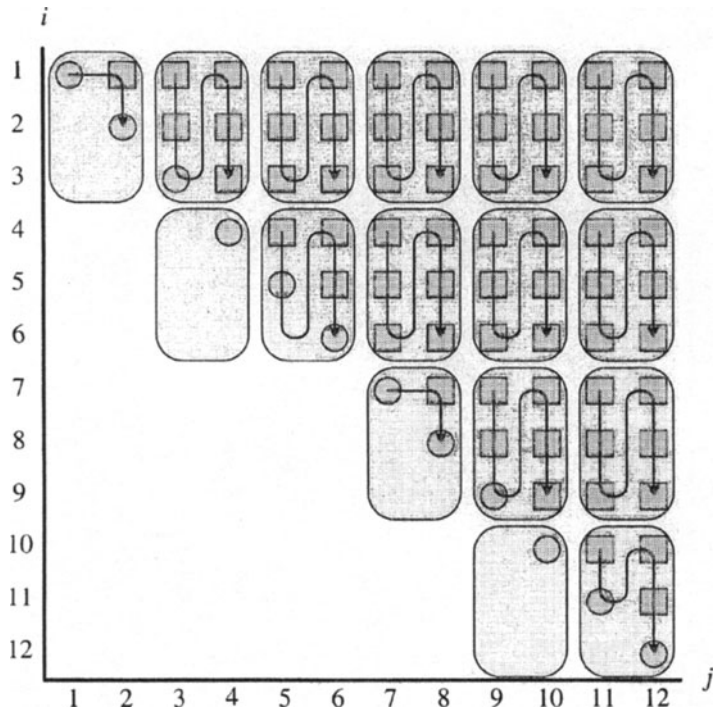


Figure 3.5. Ordering scheme parameterized by ρ and ψ for the SH algorithm where $\rho = 2$, $\psi = 3$, $m \geq n$, and $n = 12$.

For some high performance computer architectures, it may be necessary to manually unravel the computations that comprise the superscalar primitives, that is to hard code the register and compiler parameterizations (Andersson et al., 1998). If the compiler is not capable of unraveling the component computations of the superscalar primitives, then the compiler will quickly run out of registers before even a handful of primitives are executed.

Fortunately, optimal settings for the Superscalar Parameterization depend solely on the size of the register file, the number of scalar units, and the optimizing compiler – factors that are invariant to problem size.

2.2 Memory Hierarchy Parameterization

The second layer of parameterization focuses on reducing data traffic between node memory and the caches. Parameters are introduced to control *temporal reuse*. Temporal reuse occurs when multiple accesses to a single memory element occur close enough in time such that the element still resides

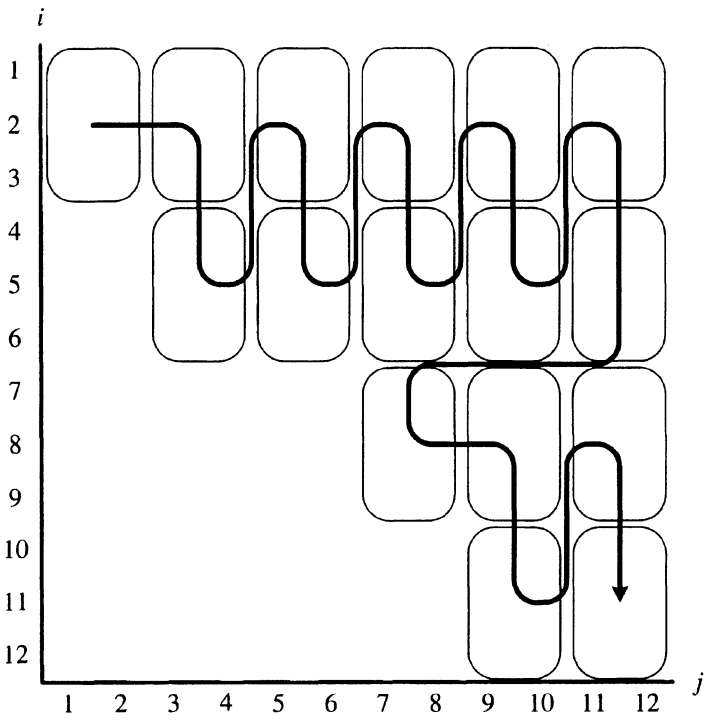


Figure 3.6. Ordering scheme parameterized by h for the SH algorithm where $h = 2$, $\rho = 2$, $\psi = 3$, $m \geq n$, and $n = 12$.

in the caches. Temporal reuse is improved by introducing the cache parameter h to control the ordering of the superscalar primitives using the following procedure:

Procedure: Memory Hierarchy Parameterization

Step 1: Select a single source superscalar primitive and denote it by the symbol s_1

Step 2: Assume only the values needed to execute s_1 are stored in the cache

Step 3: Set $i = 1$

Step 4: Let V_i be the set of all primitives not in the set $\{s_1, s_2, \ldots, s_i\}$ that share a path of length 1, or share no path of any length with the set of primitives $\{s_1, s_2, \ldots, s_i\}$

Step 5: Let S_i be a subset of V_i where a primitive in V_i is also in S_i, if and only if all of its predecessor primitives that share a path length of 1 are in the set $\{s1, s2, \ldots, si\}$

Step 6: For each and every primitive in S_i, tabulate any additional data that will have to be loaded into the cache before the primitive can be executed

Step 7: Select the primitive from S_i that requires the largest number of additional loads/stores and denote it by the symbol $s_i + 1$

Step 8: Let $i = i + 1$

Step 9: Repeat steps 4-8 $(h - 1)$ times where h is the cache parameter

Step 10: Set $k = i$

Step 11: Assume the cache is large enough to store only the values needed to execute $\{s_k - h + 1, s_k - h + 2, \ldots, s_k\}$

Step 12: Select the primitive from s_i that requires the smallest number of additional loads/stores and denote it by the symbol s_{i+1}

Step 13: Let $i = i + 1$

Step 14: Repeat steps 11-16 until the primitive s_i no longer reuses any stored values in the cache

Step 15: Return to step 6 while V_i is non-empty

For example, the Memory Hierarchy Parameterization procedure applied to the SH algorithm produces the ordering in Figure 3.7.

The parameterizations introduced so far can be used in single processor implementations. The resulting sequential, parameterized algorithm can be used to probe the performance characteristics of a single superscalar processor.

2.3 Multiprocessor Parameterization

The final layer of parameterization focuses on the problem of partitioning the computational work to reduce *load imbalance* among the processing elements. The multiprocessor parameter w aggregates superscalar primitives into tasks. The aggregation of superscalar primitives should complement the ordering parameterized by h. Much like the superscalar parameter ρ, the parameter w cuts the ordering scheme parameterized by h, ρ, and ψ into segments. For example, the Multiprocessor Parameterization applied to the SH algorithm produces the partitioning scheme in Figure 3.7.

Tasks that can be computed concurrently comprise a *concurrency set*. Each and every task must belong to one and only one concurrency set. The problem

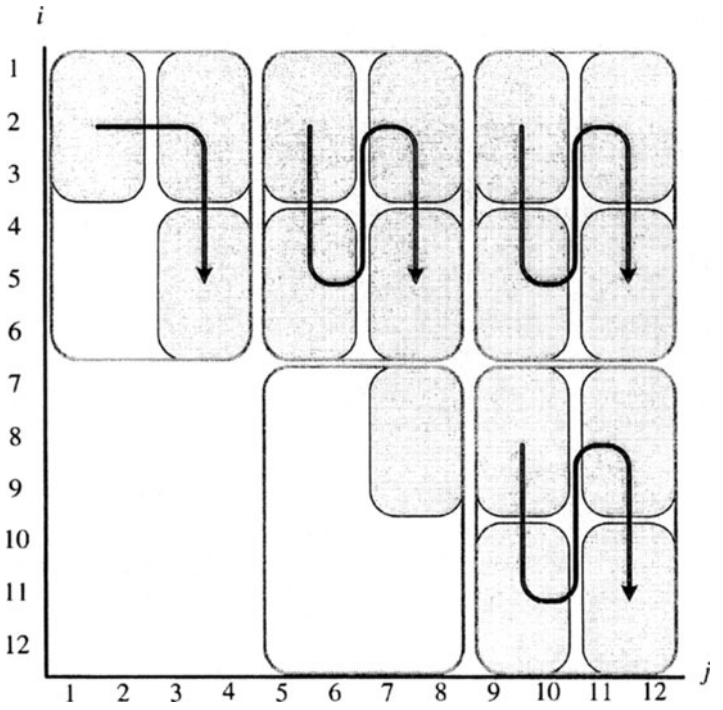


Figure 3.7. Ordering scheme parameterized by w for the SH algorithm where $w = 2$, $h = 2$, $\rho = 2$, $\psi = 3$, $m \geq n$, and $n = 12$.

of assigning all tasks to multiple processors is reduced to only assigning tasks within a single concurrency set to multiple processors. By computing the total computational work within a concurrency set, tasks can be assigned to P processors in such a manner as to distribute the computational work as evenly as possible. Tasks are non-preemptive – once initiated, a task executes to completion.

Depending on the underlying dependencies among rotations, the composition of the concurrency sets may not be unique. If possible, final selection of the concurrency sets should depend on profiling data.

3. Related Work

In the last couple of years, automatic techniques for exploiting various aspects of parallel architectures have been the topic of much research (Darte and Vivien, 1997; Lim and Lam, 1998; Sarkar, 1997; Smith and Suri, 2000; Bodin and O'Boyle, 1996; Wolf and Lam, 1991; Wolfe, 1996b; Wolfe, 1996a).

Wolf and Lam, for instance, present an algorithm for handling combinations of loop reversal, loop skewing, and loop interchange to improve parallelism. The algorithm first detects all fully permutable loops – the loops can be transformed such that there are no dependent cycles between iterations. It then transforms a set of d fully permutable loops into one sequential inner loop and $d-1$ outer loops with independent iterations or *parallel loops*. Unfortunately, the optimality of this approach is limited to single-statement loop nests (Darte and Vivien, 1997). In a more recent paper, Lim and Lam propose an algorithm that minimizes the number of synchronizations for k degrees of parallelism where k specifies the desired number of parallel loops. The idea is to choose k such that a sufficient amount of parallelism can be found to fully occupy the target architecture. Their technique encompasses the transformations of Wolf and Lam and adds loop fusion, loop fission, and loop scaling, among others. The success of the approach hinges on the assumption that the amount of data communication between processors is directly proportional to the number of synchronizations. Hence, by minimizing the number of synchronizations, they hope to minimize data communication and in turn minimize execution time.

Darte and Vivien have developed a technique for maximizing the number of loops with independent iterations in a nested set of loops. The number of loops generated by their approach is a function of how complex the dependencies are among computations. Darte and Vivien use an inexact representation of the dependencies among computations and cannot take full advantage of the freedom that may exist to reorder computations for efficient parallel execution. As the experimental results show in Chapter 5, neither the approach of Darte and Vivien nor the approach of Lim and Lam necessarily results in the best execution time.

Much of Lam and her students' work has been incorporated into the Stanford University Intermediate Format (SUIF) compiler (Wilson et al., 1994). SUIF is a powerful research platform for evaluating new developments in the optimizing compiler community. Algorithm designers to gain insight into how dependencies among computations can be manipulated to expose parallelism can also use the compiler.

While a compiler optimization may decrease the number of synchronizations or increase the amount of exploitable parallelism, it may also degrade the performance of the memory system or stall the instruction pipelines. The successive application of a handful of compiler optimizations can produce multiprocessor programs that run even slower than their sequential counterparts. The order in which the optimizations are applied is also very important. High-level optimizations for exposing multiprocessor parallelism may adversely affect sequential performance. The severity of the impact is proportional to the number of scalar units. As a consequence, low level optimizations, including inner-loop unrolling, and instruction scheduling should be

done before high-level optimizations are applied. This is problematic because low level optimizations are typically applied to assembly code, and high-level optimizations are typically applied to high-level language code. The Parallel Algorithm Synthesis Procedure follows the philosophy of first applying low level optimizations (superscalar parameterization) and then applying high-level optimizations (memory hierarchy and multiprocessor parameterizations). The case studies in Chapters 5, 6, and 7 showcase the efficacy of this philosophy.

Besides automatic techniques for exploiting parallel computer architectures, various authors have proposed synthesis procedures for addressing a particular aspect of parallel algorithm design. Stankovic et al. (1995) review classical scheduling results for the multiprocessor task assignment problem. While the results do not provide direct solutions, the authors advocate using the results to gain some insight in how to avoid poor scheduling algorithm choices. Gallivan et al. (1988) propose an analysis strategy for quantifying the benefits of blocking. The authors apply their analysis strategy to matrix-matrix multiplication kernels in the level-3 library of the basic linear algebra subroutine library (BLAS, Lawson et al., 1979; Dongarra et al., 1990). Robert (1990) reviews various algorithm design procedures in the context of LU decomposition.

Many of the techniques employed by the proposed synthesis procedure to partition and schedule computations and communications are closely related to compiler optimizations and algorithm synthesis strategies developed previously by other researchers. The superscalar and memory hierarchy parameterizations are for instance very closely related to a graph blocking scheme known as "tiling" (Desprez et al., 1998; Wolfe, 1996b). What differentiates the Parallel Algorithm Synthesis Procedure from other available compiler optimizations and synthesis procedures is the explicit use of parameters to control the optimizations and the order in which the parameters are introduced. The parameters permit the algorithm designer to explore the tradeoff between maximizing coarse-grain parallelism, maximizing fine-grain parallelism, minimizing communication, and minimizing the number of synchronizations. Optimizing compilers solve these interdependent problems sequentially and separately. The above tradeoff is extremely sensitive to factors that are not available at compile time, such as the dimensionality of the problem. It is difficult to select an appropriate strategy without some sort of bound on m and n. Interactive compilers that pole the algorithm designer for boundary information on some runtime parameters would be very useful in addressing this problem.

Chapter 4

REVIEW OF MATRIX FACTORIZATION

Matrix factorization algorithms lie at the heart of many signal processing applications. For example, the problem of finding a vector x, such that $Ax = b$, where A is an $m \times n$ complex matrix and b is a complex n-length vector, is a particularly important and computationally intensive problem in adaptive beamforming applications. When $m \geq n$, the problem is overdetermined – there are more equations than unknowns. In general, overdetermined systems have no exact solution, but a suitable approximation is to find a vector x that minimizes the total squared error or solves the least squares problem:

$$\min_{x \in C} ||Ax - b||_2$$

If A has full column rank, then there is a unique vector x that minimizes the least squares problem and solves the linear system $A^H Ax = A^H b$ (Golub and Van Loan, 1989).

Computing the matrix factorization $A = QR$ is the most reliable and efficient procedure for determining the least squares solution of an overdetermined system of linear equations. Rather than finding a vector x that minimizes $||Ax - b||_2$, QR factorization can be used to find a vector y that solves the equation $Ry = g$, where the matrix $R = Q^H A$ is upper triangular, $c = Q^H b$, and $Q^H Q = I$. Because of the special properties of the matrix Q, the vector y that solves the equation $Ry = c$ is also the vector that minimizes

$$||Ax - b||_2 = ||Q^H Ax - Q^H b||_2 = ||Ry - c||_2.$$

Given the matrix R and the vector c, backward substitution can exploit the upper triangular structure of R to solve the equation $Ry = c$ directly.

An example of a least squares problem is fitting a straight line to an experimentally determined set of data points. For example, consider the problem

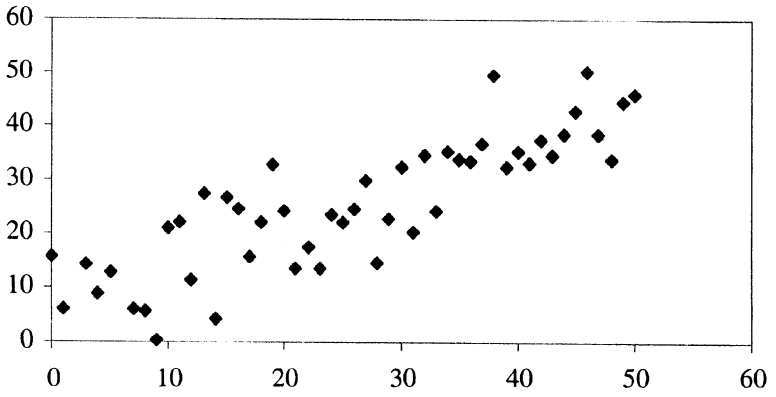


Figure 4.1. Source data for line fitting example.

of fitting a line to the data points $(a_1, b_1), (a_2, b_2), \ldots, (a_{50}, b_{50})$ in Figure 4.1. Clearly, there are no unique values of x_1 and x_2 that satisfy all of the linear equations $b_1 = x_1 + a_1 x_2, b_2 = x_1 + a_2 x_2, \ldots, b_{50} = x_1 + a_{50} x_2$. As a consequence, a suitable fit can be found by solving the corresponding least squares problem:

$$\min_{x \in \mathcal{R}} ||Ax - b||_2$$

where

$$A = \begin{bmatrix} 1 & a_1 \\ 1 & a_2 \\ \vdots & \vdots \\ 1 & a_{50} \end{bmatrix} \quad \& \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{50} \end{bmatrix}.$$

By computing the QR factorization of A and applying the transformation Q to b, the solution is straightforward for the line-fitting example:

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} \\ 0 & r_{2,2} \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix} \quad \& \; c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{50} \end{bmatrix}.$$

The resulting solution is depicted in Figure 4.2 for $x_1 = 4.55$ and $x_2 = 0.79$. While the problem of fitting a line to a data sequence is a convenient example, the special Vandermonde structure of A allows for a much simpler solution procedure that does not include QR factorization (Golub and Van Loan, 1989).

There are a number of well-known algorithms for computing the QR factorization of a matrix including Givens, Householder, and Gram-Schmidt methods. In this chapter, two Givens-based algorithms, two Householder-based

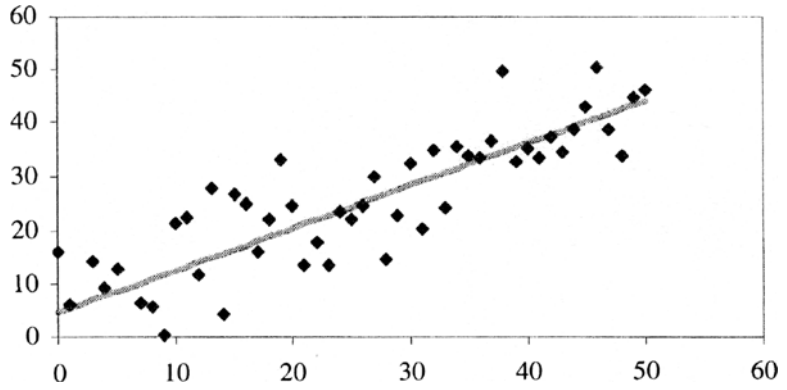


Figure 4.2. Resulting line fit using QR factorization.

algorithms, and a Householder-based matrix bidiagonalization algorithm are discussed. The algorithms take as input complex data. However for programmatic simplicity, the case studies in Chapters 5, 6, and 7 are based on real versions of the algorithms presented in this chapter.

1. Givens-based Solution Procedures

Givens rotations can be used to selectively zero elements of a matrix. In this section, two types of Givens rotations are discussed: standard and fast.

1.1 Standard Givens QR Factorization

A Givens rotation is an orthogonal reflection in $\mathcal{C}^{2\times 2}$ and has the form

$$G(i,k,j) = \begin{bmatrix} c(i,k,j) & s(i,k,j) \\ -s(i,k,j)^* & c(i,k,j) \end{bmatrix}. \tag{4.1}$$

If

$$X = \begin{bmatrix} & \vdots & \vdots & \\ \cdots & x_{k,j} & x_{k,j+1} & \cdots \\ & \vdots & \vdots & \\ \cdots & x_{i,j} & x_{i,j+1} & \cdots \\ & \vdots & \vdots & \end{bmatrix},$$

$$x_{k,j} \neq 0,$$

and

$$r(i,k,j) = \sqrt{|x_{k,j}|^2 + |x_{i,j}|^2} \tag{4.2}$$

$$c(i,k,j) = \frac{|x_{k,j}|}{r(i,k,j)} \tag{4.3}$$

$$s(i,k,j) = \frac{-c(i,k,j)\, x^*_{i,j}}{x^*_{k,j}} \tag{4.4}$$

then

$$\left[G(i,i-1,j)^H X_{i-1:i,j:j+1}\right]_{i-1:i,j} = \begin{bmatrix} x_{i-1,i-1}\, r(i,i-1,j)/|x_{i-1,i-1}| \\ 0 \end{bmatrix},$$

$$\left[G(i,i-1,j)^H X_{i-1:i,j:j+1}\right]_{i-1:i,j+1} = \begin{bmatrix} c(i,i-1,j)\, x_{i-1,j+1} - s(i,i-1,j)\, x_{i,j+1} \\ s(i,i-1,j)^*\, x_{i-1,j+1} + c(i,i-1,j)\, x_{i,j+1} \end{bmatrix},$$

and

$$G(i,i-1,j)^H G(i,i-1,j) = \begin{bmatrix} c(i,i-1,j)^2 + c(i,i-1,j)^2 |x_{i,j}|^2/|x_{i-1,j}|^2 & 0 \\ 0 & c(i,i-1,j)^2 + c(i,i-1,j)^2 |x_{i,j}|^2/|x_{i-1,j}|^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The Standard Givens (SG) QR factorization algorithm applies a sequence of Givens rotations to annihilate the subdiagonal elements of a complex $m \times n$.

Algorithm: SG (Standard Givens QR Factorization)

Input(A)

$A^0 = A$

$[m,n] = \text{dimensions}(A^0)$

$k = 0$

For $j = 1$ to $\min(m-1, n)$

For $i = m$ to $j+1$ by -1

If $A^k_{i-1,j} \neq 0$ then use Eqs. 4.2 – 4.4 to compute $G(i,i-1,j)$

$A^{k+1}_{i-1:i,j:n} = G(i,i-1,j)^H A^k_{i-1:i,j:n}$

$k = k+1$

End If

End For

End For

$A = A^k$

Output(A)

The annihilation proceeds from bottom to top within each column and then from left to right, column-by-column. Explicit computation of the matrix Q is not often necessary in signal processing applications where QR is performed. As a consequence, high-level descriptions of many QR factorization algorithms do not include the steps necessary to determine Q, and this includes the algorithms presented in this chapter.

1.2 Fast Givens QR Factorization

Standard Givens rotations are inefficient on computer architectures capable of performing one or more multiply-accumulates per clock cycle. At the heart of a Givens rotation are four multiplications and two additions. Fast Givens rotation is comprised of two multiplications and two additions. While fast Givens rotations are not orthogonal, they can be used to solve least squares problems.

The Standard Fast Givens (SFG) QR factorization algorithm applies a sequence of fast Givens rotations to reduce a real $m \times n$ matrix A to upper triangular. The rotations have the following forms:

$$F_1(i,k,j) = \begin{bmatrix} \beta_1(i,k,j)^* & 1 \\ 1 & \alpha_1(i,k,j)^* \end{bmatrix}$$

for a "type 1" rotation and

$$F_2(i,k,j) = \begin{bmatrix} 1 & \alpha_2(i,k,j)^* \\ \beta_2(i,k,j)^* & 1 \end{bmatrix}$$

for a "type 2" rotation.

If

$$X = \begin{bmatrix} & \vdots & \vdots & \\ \cdots & x_{k,j} & x_{k,j+1} & \cdots \\ & \vdots & \vdots & \\ \cdots & x_{i,j} & x_{i,j+1} & \cdots \\ & \vdots & \vdots & \end{bmatrix},$$

$x_{i,j} \neq 0,$

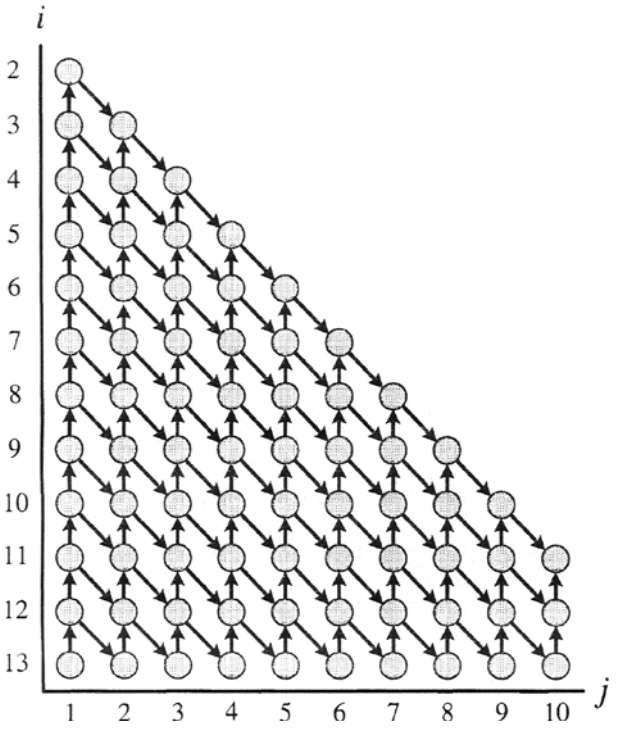


Figure 4.3. Dependency graph for the case $m = 13$ and $n = 10$.

$$x_{k,j} \neq 0,$$

$$\begin{aligned} \alpha_1(i,k,j) &= -x_{i,j}^* \, x_{k,j} \, / \, |x_{i,j}|^2 && (4.5)\\ \beta_1(i,k,j) &= -\alpha_1(i,k,j)^* \, D_{i,i} \, / \, D_{k,k} && (4.6)\\ \gamma_1(i,k,j) &= \text{real}(-\alpha_1(i,k,j) \, \beta_1(i,k,j)), && (4.7) \end{aligned}$$

and

$$\begin{aligned} \alpha_2(i,k,j) &= -x_{k,j}^* \, x_{i,j} \, / \, |x_{k,j}|^2 && (4.8)\\ \beta_2(i,k,j) &= -\alpha_2(i,k,j)^* \, D_{k,k} \, / \, D_{i,i} && (4.9)\\ \gamma_2(i,k,j) &= \text{real}(-\alpha_2(i,k,j) \, \beta_2(i,k,j)) && (4.10) \end{aligned}$$

then

$$F_1(i,i-1,j)^H X_{i-1:i,j:j+1}$$

$$= \begin{bmatrix} \beta_1(i,i-1,j)\, x_{i-1,j} + x_{i,j} & \beta_1(i,i-1,j)\, x_{i-1,j+1} + x_{i,j+1} \\ 0 & x_{i-1,j+1} + \alpha_1(i,i-1,j)\, x_{i,j+1} \end{bmatrix}$$

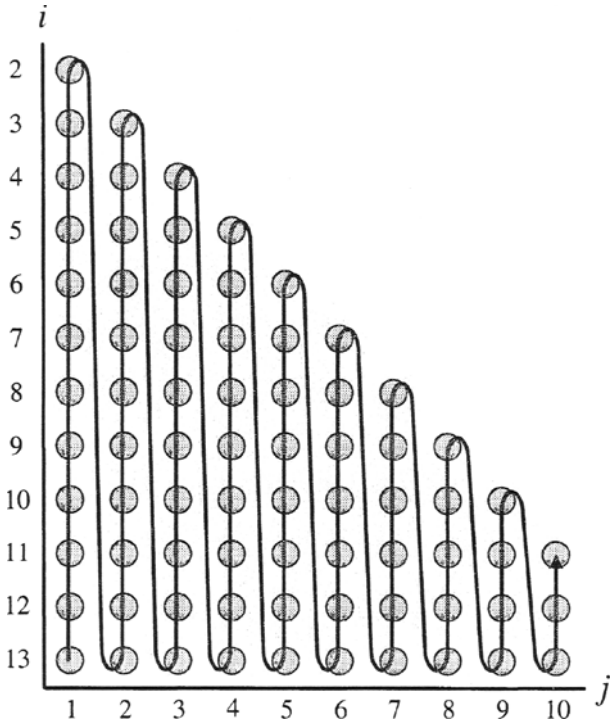


Figure 4.4. SFG ordering for the case $m = 13$ and $n = 10$.

for a "type 1" fast Givens rotation and

$$F_2(i, i-1, j)^H X_{i-1:i,j:j+1} = \begin{bmatrix} x_{i-1,j} + \beta_2(i, i-1, j)\, x_{i,j} & x_{i-1,j+1} + \beta_2(i, i-1, j)\, x_{i,j+1} \\ 0 & \alpha_2(i, i-1, j)\, x_{i-1,j+1} + x_{i,j+1} \end{bmatrix}$$

for a "type 2" fast Givens rotation. The appropriate type of rotation is chosen to minimize the growth in the entries of D and X. The algorithm is presented below:

Algorithm: SFG (Standard Fast Givens QR Factorization)

Input(A)

$A^0 = A$

$[m, n] = \text{dimensions}(A^0)$

$k = 0$

For $j = 1$ to $\min(m-1, n)$

For $i = m$ to $j+1$ by -1

If $A^k_{i,j} \neq 0$ and $A^k_{i-1,j} \neq 0$ then use Eqs. 4.5 - 4.10 to compute parameters α_1, α_2, β_1, β_2, γ_1, and γ_2

If $\gamma_1(i, i-1, k) < 1$ then $A^{k+1}_{i-1:i,j:n} = F_1(i, i-1, j)^H A^k_{i-1:i,j:n}$ and $D^{k+1} = \gamma_1(i, i-1, j) D^k$

Else $A^{k+1}_{i-1:i,j:n} = F_2(i, i-1, j)^H A^k_{i-1:i,j:n}$ and $D^{k+1} = \gamma_2(i, i-1, j) D^k$

$k = k+1$

End If

End For

End For

$A = A^k$

Output(A).

Let $r_{i,j}$ denote the application of a rotation to introduce a zero in row i of column j by combining rows i and $i-1$. Figure 4.4 shows the SFG rotation ordering in the case $m = 13$ and $n = 10$ where each shaded circle represents $r_{i,j}$. No element of the matrix A is annihilated more than once, and since row i was last modified in $r_{i+1,j}$ and row $i-1$ was last modified in $r_{i-1,j-1}$, $r_{i,j}$ depends on $r_{i+1,j}$ and $r_{i-1,j-1}$. From these dependency relationships, a dependency graph is derived as depicted in Figure 4.3 for the case $m = 13$ and $n = 10$. Any path through the graph that does not violate the dependencies and traverses each rotation once and only once will possess the same numerical properties as the SFG algorithm. These dependency relationships and the corresponding dependency graph can also be used to describe the SG algorithm.

2. Householder-based Solution Procedures

Central to the Householder-based solution procedures are Householder reflections. If $w \in \mathcal{C}^{m \times 1}$, $w^H w > 0$, and $\tau = -2/(w^H w)$, then a Householder reflection is defined as a matrix $H = I + \tau w w^H$. A Householder reflection can selectively zero elements of a vector. This capacity, in conjunction with the fact that a Householder reflection is orthogonal,

$$\begin{aligned} H^H H &= (I + \tau w w^H)^H (I + \tau w w^H) \\ &= I - 4 w w^H \frac{4}{w^H w} + \frac{4}{(w^H w)^2} w (w^H w) w^H \\ &= I \end{aligned}$$

is what makes Householder reflections particularly useful in solving least squares problems.

2.1 Householder QR Factorization

The standard Householder QR factorization (SH) algorithm applies a sequence of $\min(m-1, n)$ Householder reflections to reduce a complex $m \times n$ matrix A to upper triangular form. The reduction proceeds one column at a time from left-to-right using the following algorithm:

Algorithm: SH (Standard Householder QR Factorization)

Input(A)

$A^0 = A$

$[m, n] = \text{dimensions}(A^0)$

For $k = 1$ to $\min(m-1, n)$

$\beta = -\text{sign}(\text{real}(A_{k,k}^{k-1}))||A_{k:m,k}^{k-1}||_2$

$\alpha = 1/(A_{k,k}^{k-1} - \beta)$

$\tau^k = (\beta - A_{k,k}^{k-1})/\beta$

$v_1^k = 1$

$v_{2:m-k+1}^k = A_{k+1:m}^{k-1}/\alpha$

$A_{k:m,k:n}^k = (I - \tau^{k*} v^k v^{k\,H})^H A_{k:m,k:n}^{k-1}$

End For

$A = A^k$

Output(A).

The computation of α, β, τ^i, and w^i is denoted by the symbol l_i and the computations necessary to apply the H^i to the j^{th} column of A^{i-1} by the symbol $s_{i,j}$. For the SH algorithm, $s_{i,j}$ depends on $s_{i-1,j}$ and on l_i. In turn, l_i depends on $s_{i-1,i}$. Figure 4.5 shows the corresponding dependency graph for the case $m \geq n$ and $n = 12$ where a circle represents l_i and a square represents $s_{i,j}$ for $j = 1, 2, \ldots, i$ and $i = 1, 2, \ldots, \min(m-1, n)$. Figure 4.6 shows the SH algorithm ordering through the graph.

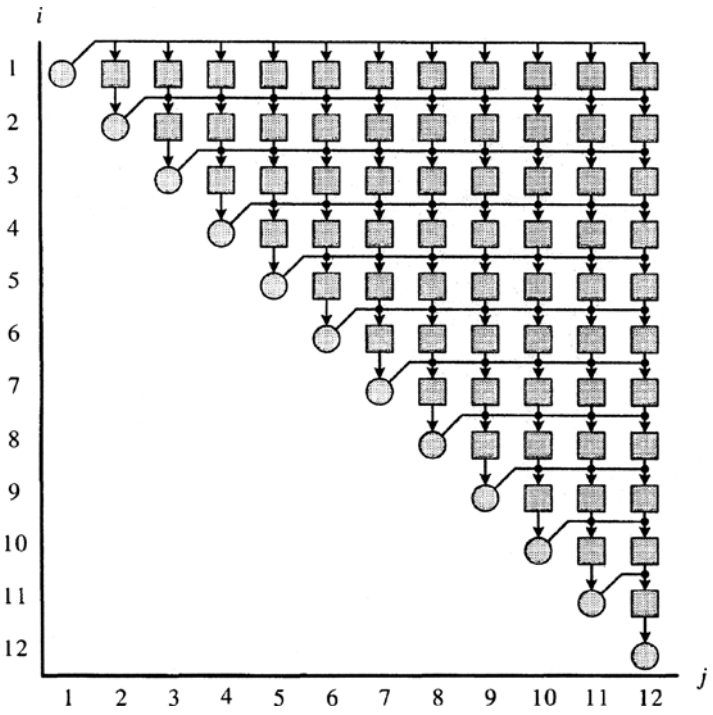


Figure 4.5. SH dependency graph for the case $m \geq n$ and $n = 12$.

2.2 Compact WY: Block QR Factorization

Bischof and Van Loan (1987) proposed a means of aggregating q Householder reflections. They showed that the product $Q = H_1 H_2 \ldots H_q$ can be written in the form $I + WY$ (the so-called "WY" representation) where $W \in \mathcal{C}^{m \times q}$ and $Y \in \mathcal{C}^{q \times m}$. Aggregation allows for the reflections to be applied in block fashion using matrix-matrix multiplication. Schreiber and Van Loan (1989) proposed a more efficient representation for the product

$$Q = I + YTY^H$$

or the Compact WY presentation where $Y \in \mathcal{C}^{m \times q}$ and $T \in \mathcal{C}^{h \times q}$. Given v_i for $i = 1, 2, \ldots, q$, the following procedure computes Y and T:

Algorithm: CWY (Compact WY)

Input$(m, \tau^1, \tau^2, \ldots, \tau^q, v^1, v^2, \ldots, v^q)$

for $i = 1$ to q

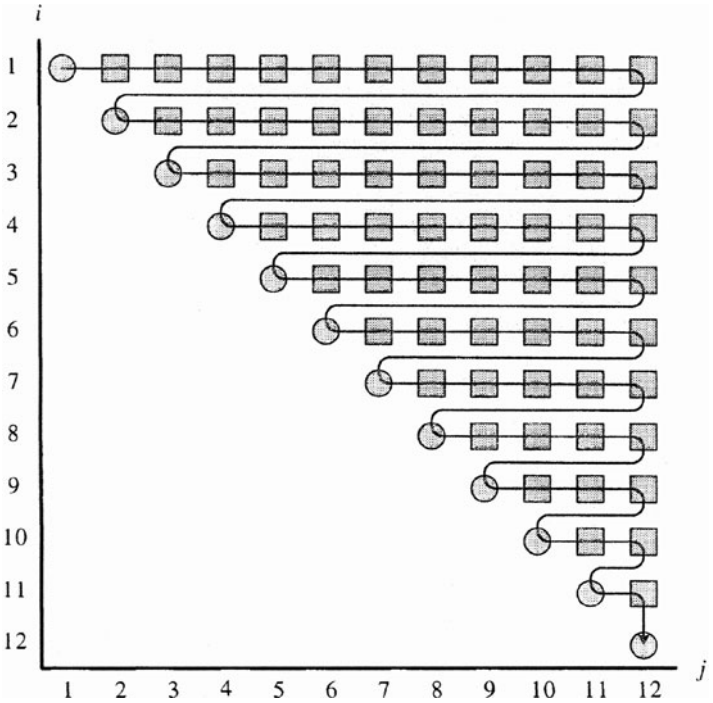


Figure 4.6. SH ordering for the case $m \geq n$ and $n = 12$.

$Y_{1:m,i} = v^i$

$T_{1:i-1,i} = -\tau^i \, T_{1:i-1,1:i-1} \, Y^H_{i:m,1:i-1} \, v^i$

$T_{i,i} = \tau_i$

End For

$T = T^q$

$T = T^q$

Output(T, Y)

If $h \ll m$, then the added computational cost associated with the Compact WY representation is negligible in comparison to the potential performance benefits of introducing matrix-matrix multiplication. The problem of computing the QR factorization of a matrix using the Compact WY representation is now straightforward:

Algorithm: SCWY (Standard Compact WY QR Factorization)

Input(A, q)

$A^0 = A$

$[m, n] = \text{dimensions}(A^0)$

$m_n = \min(m-1, n)$

For $k = 1$ to m_n by q

$\hat{n} = \min(k + 2q - 1, m_n)$

Compute $[v_k, v_{k+1}, \ldots, v_{\hat{n}}]$ using the SH algorithm to factor $A^{k-1}_{k:m,k:\hat{n}}$

Compute Y and T using the CWY algorithm

$A^{k+h}_{k:m,k+q:n} = A^k_{k:m,k+q:n} + YT^H Y^H A^k_{k:m,k+q:n}$

End For

$A = A^{k+q}$

Output(A)

2.3 Householder Bidiagonal Factorization

The Standard Householder Bidiagonal Factorization algorithm (SHB, Golub and Kahan, 1965) algorithm applies an alternating sequence of left and right Householder reflections to reduce A to the bidiagonal matrix B. More precisely, beginning with $B^0 = A$,

$$\tilde{B}^{k+1} = \left(I - \tau^{k\,*} u^k u^{k\,H}\right) B^k \tag{4.11}$$

and

$$B^{k+1} = \tilde{B}^{k+1} \left(I - \sigma^{k*} v^{k\,H} v^k\right) \tag{4.12}$$

are computed for $k = 1, \ldots, m_n$ where $m_n = \min(m-1, n)$. In Eq. (4.11), the scalar τ^k and the m-element Householder column vector u^k are determined such that the k^{th} column of $B^{k+1\,*}$ is zero below the diagonal using the SH algorithm. Similarly, in Eq. (4.12) the scalar σ^k and the n-element Householder row vector v^k are determined such that the k^{th} row of B^{k+1} is zero to the right of the superdiagonal using a row-oriented version of the SH algorithm. By taking advantage of the zeros in u^k and v^k and distributing the multiplications in Eqs. (4.11) and (4.12), the SBH algorithm may be implemented using $4n^2\,(m - n/3)$ flops.

Chapter 5

CASE STUDY 1: PARALLEL FAST GIVENS QR FACTORIZATION

Although parallel QR factorization has been the topic of much research, available parallel algorithms exhibit poor scalability characteristics on matrices with dimensions less than 3000. As a consequence, there is little flexibility to meet stringent latency constraints by manipulating the number of processors. This is particularly true of parallel algorithms based on block cyclic distribution schemes such as ScaLAPACK's PDGEQRF (Choi et al., 1995; Blackford et al., 1997). Further compounding the problem of scalability is the fact that block cyclic distribution schemes are often not compatible with the data movement patterns of many applications. Note that some very recent work on efficient real time redistribution techniques promises to make these algorithms more attractive to high performance signal processing applications (Park et al., 1999; Petit and Dongarra, 1999).

This chapter discusses the design, implementation, and performance of a parameterized, parallel fast Givens algorithm (Dunn and Meyer, 2002) for computing the factorization $A = QR$. This algorithm is well suited to signal processing applications and applies fast Givens rotations in block fashion using a strategy that is similar to the one developed by Carrig and Meyer (1999) for sequential QR factorization. Using the Parallel Algorithm Synthesis Procedure, superscalar, memory hierarchy, and multiprocessor parameters are introduced.

Despite the wealth of research on parallel algorithms, there is little consensus on which of the parallel programming environments – shared memory or message passing – consistently delivers singular levels of performance across a variety of problem dimensions and parallel computer architectures. The parallel fast Givens algorithm is implemented in shared memory, message passing, or hybrid shared memory/message passing. The hybrid environment allows interpolation between the shared memory and message passing programming

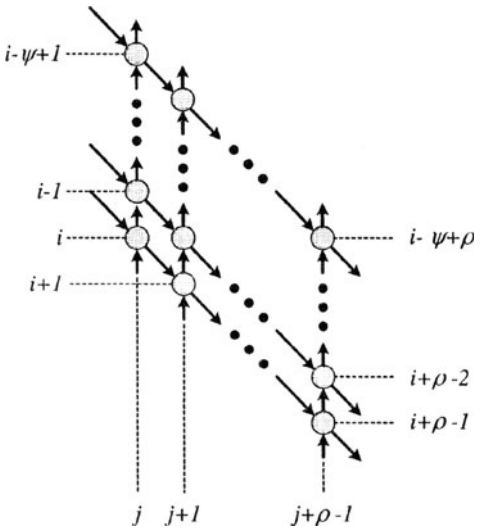


Figure 5.1. Dependency graph for a group of rotations parameterized by superscalar parameters ψ and ρ.

environments. Numerical experiments for a shared memory version and two message passing versions are conducted on a 64-processor HP SPP-2000 and a 128-processor SGI Origin 2000. For the hybrid version, a 32-processor IBM SP3 is added.

1. The Parallel Fast Givens Algorithm

This section presents the Parameterized Parallel Fast Givens (PFG) algorithm. The algorithm can be manipulated to accommodate the performance characteristics of various parallel architectures. The Parallel Algorithm Synthesis Procedure guides the introduction of the parameters.

1.1 Superscalar Parameterization

In the SFG algorithm presented in Chapter 4, the application of rotations in Figure 5.1 can require as many as $4\psi\rho n$ distinct register load and store operations for $\psi + 1 < i < m - \rho + 1$, $1 \leq \psi \ll m$, $1 \leq \rho \ll n$, and $1 \leq j \ll n - \rho + 1$. The superscalar parameterization procedure is applied to these rotations to develop a register efficient ordering. The results are shown in Figure 5.2. Steps 1 through 9 produce the vertical ordering of the rotations,

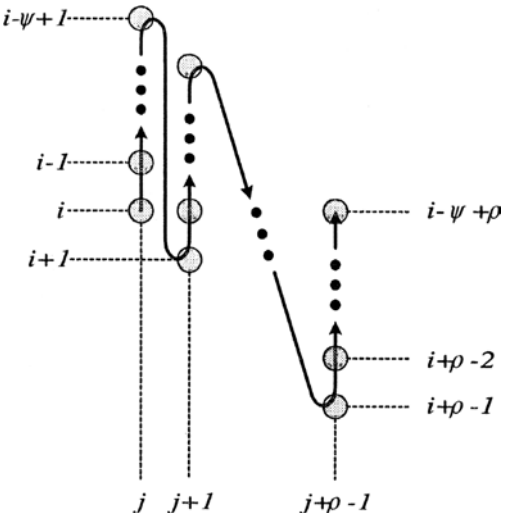


Figure 5.2. Parameterized superscalar ordering of the rotations.

and steps 10 through 15 produces the gross diagonal ordering of groups of rotations. Grouping rotations does not guarantee a register efficient ordering. However, by interleaving the computations associated with the ordering, the elements of the matrix in rows $i-\psi, i-\psi+1, \ldots, i+\rho-1$ of column k can be loaded into registers once and multiple rotations can be applied for $k = j+\rho, j+\rho+1, \ldots, n$ requiring approximately $2(\psi+\rho)n$ register loads and stores. This interleaving is accomplished by dividing the computations into two groups. The first group is comprised of only those computations necessary to determine the rotation coefficients

$$(\alpha_{t_1}^{(1)}, \beta_{t_1}^{(1)}), (\alpha_{t_2}^{(2)}, \beta_{t_2}^{(2)}), \ldots, (\alpha_{t_{\psi\rho}}^{(\psi\rho)}, \beta_{t_{\psi\rho}}^{(\psi\rho)})$$

where $t_i \in \{1,2\}$ for $i = 1, 2, \ldots, \psi\rho$ describes whether the rotation is of type 1 or type 2. The rotation coefficients are computed in the order in which they are enumerated, and this corresponds to the superscalar ordering shown in Figure 5.2. The total number of computations for this group is approximately $\psi\rho(2\rho+18)$. The second group is comprised of roughly the remaining $4\psi\rho n$ computations. These computations are involved in applying the rotation coefficients to columns $j+\rho-1, j+\rho, \ldots, n$ of the matrix A. By applying the coefficients one after another to matrix elements stored in

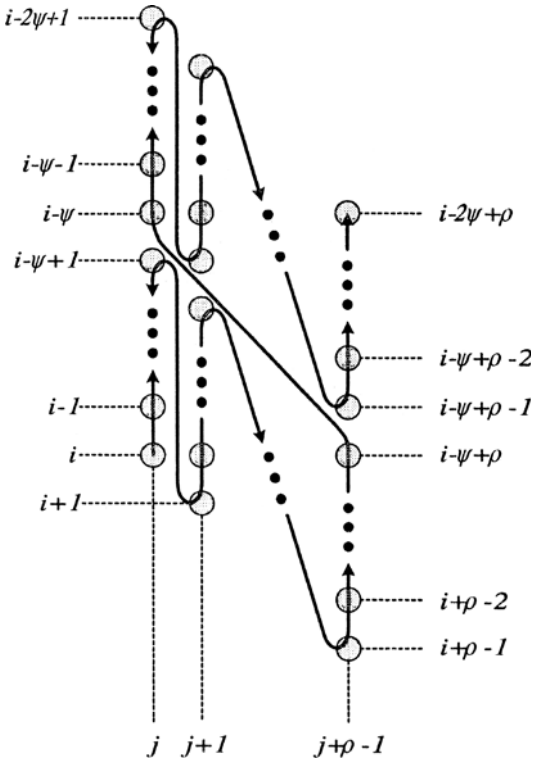


Figure 5.3. Two adjoining groups of rotations parameterized by the superscalar parameters ψ and ρ.

the register bank, the number of register load and store operations is reduced. This eliminates the need for intermediate storage of the matrix elements after each of the $\psi\rho$ rotations. The range of values ψ and ρ can take on is limited by the number of available registers and the problem dimensions m and n.

1.2 Memory Hierarchy Parameterization

The efficacy of the superscalar parameterization depends on the capacity of the caches to move data in and out of the registers in a timely fashion. To enhance this capacity, the memory hierarchy parameterization procedure is applied to tailor the amount of reuse among groups of $\psi\rho$ rotations to the sizes of the L-2 and L-1 caches, respectively. To simplify the discussion, let $\psi = \rho = 1$. This effectively disables the superscalar parameterization and allows discussion of the parameterization in terms of rotations instead of blocks of $\psi\rho$ rotations.

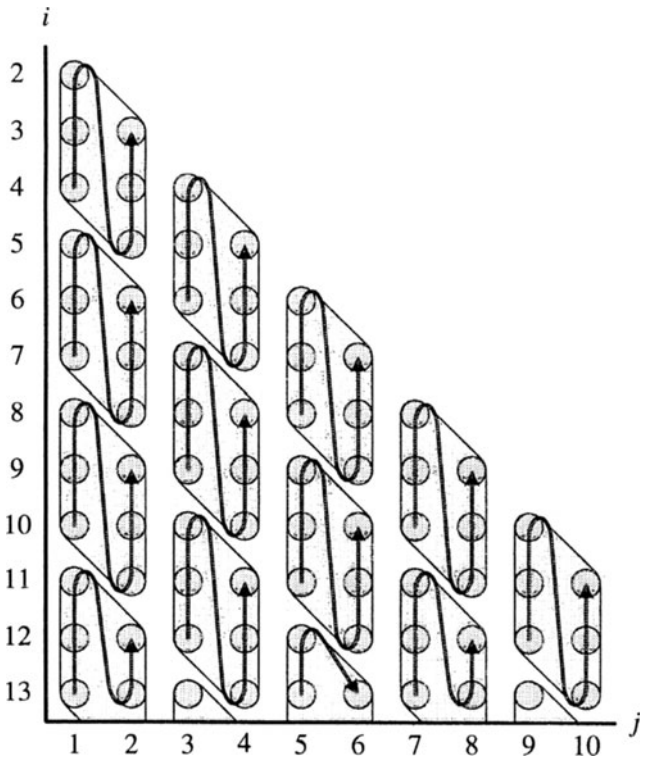


Figure 5.4. Superscalar parameterization and ordering within a superscalar block for the case $m = 13$, $n = 10$, $\psi = 3$, and $\rho = 2$.

The execution of rotations

$$R_{i-h+1,j}, R_{i-h+2,j}, \ldots, R_{i,j}$$

uses rows $i-h, i-h+1, \ldots, i$ for $h \leq i \leq m-n+1$ and $j \leq n$. The subsequent execution of rotations

$$R_{i-h+2,j+1}, R_{i-h+3,j+1}, \ldots, R_{i+1,j+1}$$

reuses rows $i-h+1, i-h+2, \ldots, i$ and requires additionally row $i+1$. If both caches can store at least $h+2$ rows, then elements in rows $i-h+1, i-h+2, \ldots, i$ do not have to be retrieved from global memory before executing the second group of rotations in column $j+1$. The rows are already stored in the cache. Elements are loaded into the caches as much as h fewer times, or more generally for $\psi > 1$, $h\psi$ fewer times. The resulting ordering is shown in Figure 5.5 and is essentially a scaled version of the superscalar parameterization.

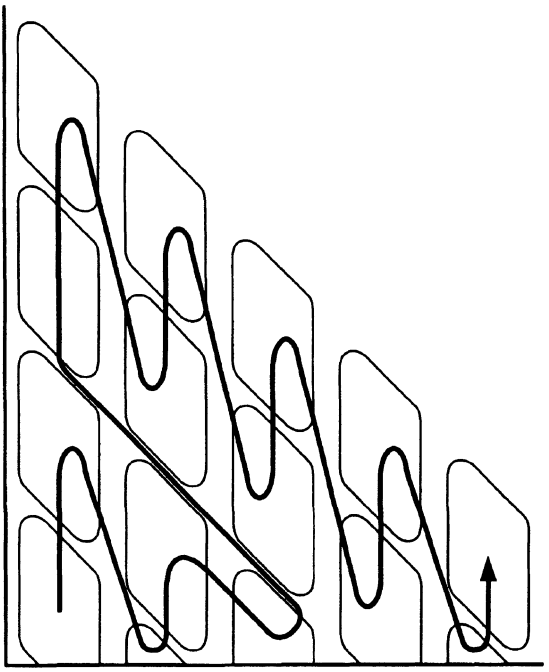


Figure 5.5. Memory hierarchy parameterization and ordering for the case $m = 13$, $n = 10$, $h = 2$, $\psi = 3$, and $\rho = 2$.

The second cache parameter d becomes necessary because the parameter h is not likely to simultaneously satisfy the larger L-2 and the smaller L-1 caches. The L-2 cache is typically multiple orders of magnitude larger than the L-1 cache. If the L-2 cache is just large enough to store $(h\psi + \rho)n$ elements, or roughly the number of elements involved in the execution of $h\psi\rho$ rotations, then the problem of improving reuse in the L-1 cache entails decoupling its storage capacity from the width of matrix n. This decoupling is accomplished by introducing a second cache parameter d such that the L-1 cache must store at least $(2\psi + \rho)d$ elements. Two adjoining groups of rotations with the same set of column indices, such as the ones depicted in Figure 5.3, share ρ rows of data. The cache parameter d breaks the associated computations that are involved in applying the rotation coefficients to columns $j + \rho - 1, j + \rho, \ldots, n$ into groups that operate on d columns or $(2\psi + \rho)d$ elements at a time. Of the $(2\psi + \rho)d$ elements stored in the L-1 cache, $d\rho$ elements are reused. Note that the cache parameter d has no influence on the

overall ordering of the rotations. It only affects the ordering of the component computations.

1.3 Multiprocessor Parameterization

In addition to defining a family of subsequential orderings through the dependency graph as depicted in Figures 5.4 and 5.5, the superscalar parameters ψ and ρ, and the memory hierarchy parameter h also define indivisible groups of rotations of size at most $h\psi\rho$. The multiprocessor parameter w is introduced to aggregate these indivisible groups of rotations into *tasks*

$$\begin{array}{c} T^1_{\tau^1_l}, T^1_{\tau^1_l+1}, \ldots, T^1_{\tau^1_r} \\ T^2_{\tau^2_l}, T^2_{\tau^2_l+1}, \ldots, T^2_{\tau^2_r} \\ \cdots \\ T^S_{\tau^S_l}, T^S_{\tau^S_l+1}, \ldots, T^S_{\tau^S_r} \end{array}$$

where

$$\tau^s_l = \max\left(1, s - \left\lceil \frac{m-1}{h\psi} \right\rceil + 1\right) \tag{5.1}$$

$$\tau^s_r = \left\lceil \frac{h\psi(s-1) + h\psi - 1}{h\psi + w\rho} + 1 \right\rceil \tag{5.2}$$

$$S = \left\lceil \frac{m-1}{h\psi} \right\rceil + \left\lceil \frac{\min(m-1, n) - w\rho}{w\rho} \right\rceil \tag{5.3}$$

for the *synchronization index* $s = 1, 2, \ldots, S$. Task T^s_τ is defined to be the set of rotations $\{R_{i,j}\}$ such that

$$\begin{aligned} i &= m - \tilde{s} + \tilde{\tau} + j - 1, m - \tilde{s} + \tilde{\tau} + j, \ldots, \min(m - \tilde{s} - h\psi + \tilde{\tau} + j, 2) \\ j &= 1 + \bar{\tau}, 2 + \bar{\tau}, \ldots, \min(\bar{\tau} + w\rho, n) \end{aligned}$$

where $\tilde{s} = h\psi(s-1)$, $\tilde{\tau} = (h\psi + w\rho)(\tau - 1)$, $\bar{\tau} = w\rho(\tau - 1)$, the *task index* $\tau = \tau^s_l, \tau^s_l + 1, \ldots, \tau^s_r$, and $s = 1, 2, \ldots, S$. Figure 5.6 depicts the synchronization and task indices for the case $m = 13$, $n = 10$, $w = 1$, $h = 2$, $\psi = 3$, and $\rho = 1$. There are no dependent cycles between tasks. As a consequence, once a processor begins applying rotations in a task, all rotations in that task are applied without any need for further synchronization. Also, from the underlying dependencies between rotations, note that tasks sharing the same synchronization index are independent and can be computed concurrently. This leads to the definition of a *concurrency set* $C_s = \{T^s_{\tau^s_l}, T^s_{\tau^s_l+1}, \ldots, T^s_{\tau^s_r}\}$ for $s = 1, 2, \ldots, S$.

Concurrency sets are executed sequentially in the order in which they are enumerated. This is sufficient to satisfy the underlying constraints among

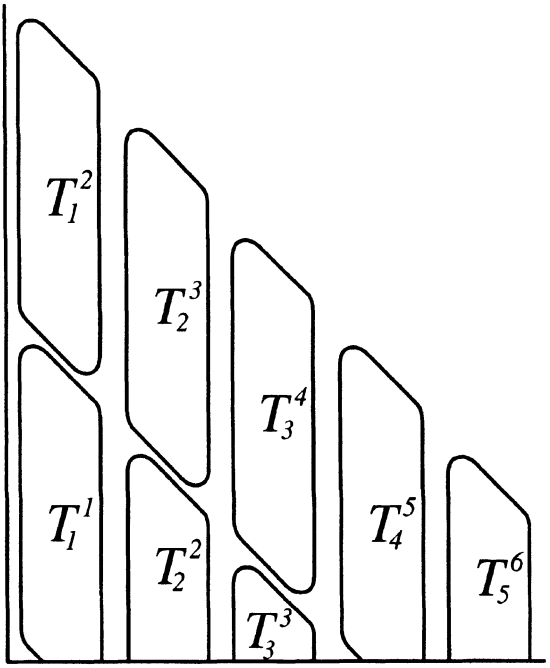


Figure 5.6. Synchronization and task indices for the case $m = 13$, $n = 10$, $w = 1$, $h = 2$, $\psi = 3$, and $\rho = 1$.

rotations and places no restriction on the order of execution for tasks within a concurrency set.

As there are no restrictions on the order of execution for tasks within a concurrency set, the computational load can be distributed across the processors as evenly as possible. This is accomplished by parceling out to processors P non-intersecting groups of tasks with roughly equal work in terms of floating-point operations. By controlling the number of rotations that comprise a task, the parameter w explores the tradeoff between improving reuse in the register bank and the caches and evenly balancing the load among P processors. The parameters h, ψ, and ρ were not explicitly designed to act as load balancing parameters. Nonetheless, they also control the number of rotations that comprise a task and thus have an impact similar to the parameter w. The range of values w, h, d, ψ, and ρ can take on is limited by the inequalities $1 \leq w\rho \leq n$, $1 \leq h\psi \leq m$, and $1 \leq d\rho \leq n$.

Before introducing the specifics of the load balancing algorithm, the following definitions are necessary: the *arithmetic complexity* of a set of tasks $\mathcal{A}(\{T^s_\tau, T^s_{\tau+1}, \ldots T^s_{\tau+k}\})$ is the total number of floating-point operations required to execute all the tasks in the set and the *task partition* Φ_s of concurrency set C_s to mean a set of $P+1$ integers $\{\phi^s_1, \phi^s_2, \ldots, \phi^s_{P+1}\}$ that satisfy the condition $\tau^s_l = \phi^s_1 \leq \phi^s_2 \leq \ldots \leq \phi^s_{P+1} = \tau^s_r + 1$. The problem of assigning tasks to processors can now be formulated as finding a task partition Φ_s of C_s such that

$$\begin{aligned} &\mathcal{A}(\{T^s_{\phi^s_1}, T^s_{\phi^s_1+1}, \ldots, T^s_{\phi^s_2-1}\}) \\ &\quad \approx \mathcal{A}(\{T^s_{\phi^s_2}, T^s_{\phi^s_2+1}, \ldots, T^s_{\phi^s_3-1}\}) \\ &\qquad \vdots \\ &\quad \approx \mathcal{A}(\{T^s_{\phi^s_P}, T^s_{\phi^s_{P+1}}, \ldots, T^s_{\phi^s_{P+1}-1}\}) \end{aligned}$$

for $s = 1, 2, \ldots, S$.

The partition Φ_s is computed in an iterative fashion. First, choose the smallest possible integer value ϕ^s_2 such that

$$\mathcal{A}(\{T^s_{\phi^s_1}, T^s_{\phi^s_1+1}, \ldots, T^s_{\phi^s_2-1}\}) \geq \frac{\mathcal{A}(C_s)}{P}.$$

The next step is to choose the smallest possible integer value ϕ^s_3 such that

$$\mathcal{A}(\{T^s_{\phi^s_2}, T^s_{\phi^s_2+1}, \ldots, T^s_{\phi^s_3-1}\}) \geq \frac{\mathcal{A}(C_s - \{T^s_{\phi^s_1}, T^s_{\phi^s_1+1}, \ldots, T^s_{\phi^s_2-1}\})}{P-1},$$

where the set notation $A - B$ denotes $A \cap B^C$. In general, the goal is to choose the smallest possible integer value ϕ^s_{p+1} such that

$$\begin{aligned} &\mathcal{A}(\{T^s_{\phi^s_p}, T^s_{\phi^s_p+1}, \ldots, T^s_{\phi^s_{p+1}-1}\}) \\ &\quad \geq \frac{\mathcal{A}(C_s - \{T^s_{\phi^s_1}, T^s_{\phi^s_1+1}, \ldots, T^s_{\phi^s_p-1}\})}{P-p+1} \end{aligned} \tag{5.4}$$

for $p = 1, 2, \ldots, P-1$. If each of the tasks $T^s_{\tau^s_l}, T^s_{\tau^s_l+1}, \ldots, T^s_{\tau^s_r-1}$ are comprised of exactly $wh\psi\rho$ rotations and that task $T^s_{\tau^s_r}$ is comprised of $h\psi$ rotations for $s = 1, 2, \ldots, S$, then

$$\mathcal{A}(\{T^s_{\phi^s_p}, T^s_{\phi^s_p+1}, \ldots T^s_{\phi^s_{p+1}-1}\}) = h\psi(14\tilde{\phi}^s + 4\hat{n}^s_p\tilde{\phi}^s - 2\tilde{\phi}^{s^2}) \tag{5.5}$$

and

$$\frac{\mathcal{A}(C_s - \{T^s_{\phi^s_1}, T^s_{\phi^s_2}, \ldots, T^s_{\phi^s_p-1}\})}{P-p+1} = h\psi\frac{(14n^s_p + 4\hat{n}^s_p n^s_p - 2n^{s^2}_p)}{P-p+1} \tag{5.6}$$

where $\hat{n}_p^s = n - (\phi_p^s - 1)w\rho$, $n_p^s = (\tau_r^s - \phi_p^s + 1)w\rho - w\rho + 1$, and $\tilde{\phi}^s = (\phi_{p+1}^s - \phi_p^s)w\rho$. By setting Eqs. 5.5 and 5.6 equal,

$$\tilde{\phi}^s = \frac{7}{2} + \hat{n}_p^s - \frac{1}{4}\sqrt{(14 + 4\hat{n}_p^s)^2 + 8n_p^s \frac{2n_p^s - 14 - 4\hat{n}_p^s}{P - p + 1}}. \tag{5.7}$$

The following algorithm computes ϕ_{p+1}^s for $p = 1, 2, \ldots, P-1$ that satisfies Eq. 5.4:

Algorithm: LB (Load Balancing)

Input$(s, P, m, n, w, h, \psi, \rho)$

Use Eqs. 5.1 and 5.2 to determine τ_l^s and τ_r^s, respectively

$\phi_1^s = \tau_l^s$

$\phi_{P+1}^s = \tau_r^s + 1$

For $p = 1$ to $P - 1$

Use Eq. 5.7 to solve for $\tilde{\phi}^s$

$\phi_{p+1}^s = \lceil \lceil \tilde{\phi}^s \rceil / (w\rho) \rceil + \phi_p^s$

End For

$\Phi^s = \{\phi_1^s, \phi_3^s, \ldots, \phi_P^s\}$

Ouput(Φ^s)

The LB algorithm computes the partitions $\Phi_1, \Phi_2, \ldots, \Phi_S$. The description of the new parameterized parallel fast Givens algorithm is straightforward.

Algorithm: PFG (Parameterized Parallel Fast Givens)

Input$(A, D, P, w, h, d, \psi, \rho)$

$[m, n]$ =dimensions(A)

Compute S from Eq. 5.3

For $s = 1$ to S

Compute Φ_s using the LB Algorithm

DO IN PARALLEL

For $p = 1$ to P

For $\tau = \phi_p^s$ to $\phi_{p+1}^s - 1$

Apply rotations in task T_τ^s

End For

End For

End For

Output(A, D)

2. Communication Procedures

For shared memory, communication is managed by the hardware and system software as a function of the real-time data requirements of the executing algorithm. Identifying the variables that need to be exchanged among processors to satisfy dependencies between concurrency sets is not necessary. This is not true for synchronous and asynchronous message passing. The algorithm explicitly controls how and when variables are exchanged among processors. This section describes the communication strategy for how and when variables are exchanged using two different message passing protocols: synchronous and asynchronous. For both protocols, explicit send and receive operations are necessary to satisfy dependencies among tasks. While the synchronous algorithm shares the same task sequencing scheme as the PFG algorithm, tasks are reordered for the asynchronous algorithm to take advantage of the architecture's ability to perform computation and communication simultaneously.

2.1 Synchronous Message Passing

For synchronous message passing, dependencies between concurrency sets are characterized in terms of rows. Processors determine the rows to exchange by computing the *row range* Γ_s for each concurrency set C_s. The row range Γ_s is a set of $P+1$ integers $\gamma_1^s, \gamma_2^s, \ldots, \gamma_{P+1}^s$ that satisfy the condition $1 = \gamma_1^s \leq \gamma_2^s \leq \ldots \leq \gamma_{P+1}^s = m+1$. Before any task in C_s can be executed on processor p, rows γ_p^s through $\gamma_{p+1}^s - 1$ must be stored in local memory. Given the task partition Φ_s of C_s, compute

$$\begin{aligned}\gamma_{p+1}^s = \min(&m - \tilde{s} + (\phi_{p+1}^s - 2)(h\psi + w\rho) \\ &- p\min(0, \phi_{p+1}^s - \phi_p^s - 1)(w\rho + 1) + w\rho, m+1)\end{aligned} \tag{5.8}$$

where $\tilde{s} = h\psi(s-1)$ for $p = 1, 2, \ldots, P-1$ and $s = 1, 2, \ldots, S$. For each Γ_s and processor $p \in \{1, 2, \ldots, P\}$, the communication strategy is defined by the following procedure:

Procedure: SP (Synchronous Message Passing)

Step 1: Compute Γ_s from Eq. 5.8

Step 2: If $s = 1$ then go to Step 13; else continue

Step 3: If $p-1$ is ODD go to Step 9; else continue

Step 4: If $p \neq 1$ and $\gamma_p^{s-1} < \gamma_p^s$ then send rows $[\gamma_p^{s-1} : \gamma_p^s - 1]$ to $p-1$

Step 5: If $p \neq P$ and $\gamma_{p+1}^{s-1} > \gamma_{p+1}^s$ then send rows $[\gamma_{r+1}^s : \gamma_{p+1}^{s-1} - 1]$ to $p+1$

Step 6: If $p \neq 1$ and $\gamma_p^s < \gamma_p^{s-1}$ then receive rows $[\gamma_p^s : \gamma_p^{s-1} - 1]$ from $p-1$

Step 7: If $p \neq P$ and $\gamma_{p+1}^s > \gamma_{p+1}^{s-1}$ then receive rows $[\gamma_{p+1}^{s-1} : \gamma_{p+1}^s - 1]$ from $p+1$

Step 8: Stop

Step 9: If $p \neq P$ and $\gamma_{p+1}^s > \gamma_{p+1}^{s-1}$ then receive rows $[\gamma_{p+1}^{s-1} : \gamma_{p+1}^s - 1]$ from $p+1$

Step 10: If $p \neq 1$ and $\gamma_p^s < \gamma_p^{s-1}$ then receive rows $[\gamma_r^s : \gamma_p^{s-1} - 1]$ from $p-1$

Step 11: If $p \neq P$ and $\gamma_{p+1}^{s-1} > \gamma_{p+1}^s$ then send rows $[\gamma_{p+1}^s : \gamma_{p+1}^{s-1} - 1]$ to $p+1$

Step 12: If $p \neq 1$ and $\gamma_p^{s-1} < \gamma_p^s$ then send rows $[\gamma_p^{s-1} : \gamma_p^s - 1]$ to $p-1$

Step 13: Stop

A new synchronous message passing version of the PFG algorithm is presented below. It includes one additional input parameter p. The parameter p is a unique integer identifying the processor.

Algorithm: SYNC (Synchronous Version of the PFG)

Input$(A, D, P, w, h, d, \psi, \rho, p)$

$[m, n]$ =dimensions(A)

Compute S from Eq. 5.3

For $s = 1$ to S

Compute Φ_s using the LB Algorithm

Compute Γ_s using Eq. 5.8

Communicate using the SP Procedure

For $\tau = \phi_p^s$ to $\phi_{p+1}^s - 1$

Apply rotations in task T_τ^s

End For

End For

Output(A, D)

2.2 Asynchronous Message Passing

The communication strategy employed for synchronous message passing has the effect of globally synchronizing all processors at the start of each concurrency set. For the asynchronous algorithm, the global, synchronous send and receive operations are separated into local, asynchronous operations among neighboring processors. Neighbors exchange the appropriate rows of the matrix asynchronously to satisfy dependencies between concurrency sets. Executing computation simultaneously with communication can hide the latency associated with these exchanges.

To accomplish the interlacing of computation and communication, dependencies between consecutive concurrency sets must be characterized in terms of tasks instead of rows. The goal is to identify those tasks with dependencies that are shared across multiple processors. By scheduling those tasks last among the group of tasks to be executed, the necessary communication can be completed in advance. In general, T_k^s depends on T_{k-1}^{s-1} and T_k^{s-1} for $k = \tau_l^s + 1, \tau_l^s + 2, \ldots, \tau_r^s$ for concurrency sets $s-1$ and s from the underlying dependencies among rotations. These dependencies only result in interprocessor communication if one or both of the tasks T_{k-1}^{s-1} and T_k^{s-1} have been assigned to a different processor than T_k^s. However, the following properties of the LB algorithm show that dependencies between concurrency sets $s-1$ and s that may result in communication for processor p are limited to tasks $T_{\phi_p^{s-1}}^{s-1}$, $T_{\phi_{p+1}^{s-1}-1}^{s-1}$, $T_{\phi_p^s}^s$, and $T_{\phi_{p+1}^s-1}^s$. More specifically, if T_{k-1}^{s-1} is assigned to processor p, T_k^{s-1} is assigned to processor $p+1$, and $k = \phi_{p+1}^{s-1} \neq \tau_l^s \neq \tau_r^s$; then T_k^s is either assigned to processor p or $p+1$, T_{k+1}^s is assigned to processor $p+1$, and $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$ for $p = 1, 2, \ldots, P-1$ and $s = 2, 3, \ldots, S$.

PROPERTY 2.1 *If* $\phi_p^s = \phi_p^{s-1}$, $\tau_r^s = \tau_r^{s-1}$, *and* $\phi_{p+1}^{s-1} \leq \tau_r^s$, *then* $\phi_{p+1}^s = \phi_{p+1}^{s-1}$.

Proof: *If* $\phi_p^s = \phi_p^{s-1}$ *and* $\tau_r^s = \tau_r^{s-1}$ *then*

$$\begin{aligned} &\mathcal{A}(C_s - \{T_{\phi_1^s}^s, T_{\phi_1^s+1}^s, \ldots, T_{\phi_p^s-1}^s\}) \\ &\quad = \mathcal{A}(C_{s-1} - \{T_{\phi_1^{s-1}}^{s-1}, T_{\phi_1^{s-1}+1}^{s-1}, \ldots, T_{\phi_p^{s-1}-1}^{s-1}\}). \end{aligned}$$

Therefore, $\phi_{p+1}^s = \phi_{p+1}^{s-1}$.

PROPERTY 2.2 *If* $\phi_p^s = \phi_p^{s-1}$ *and* $\tau_r^s = \tau_r^{s-1} + 1$, *then* $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.
Proof: *By definition,*

$$\mathcal{A}(T^s_{\phi^{s-1}_{p+1}}) \geq \frac{\mathcal{A}(T^s_{\phi^s_{P+1}-1})}{P-p+1}$$

for $p = 1, 2, \ldots, P$. *If*

$$14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 = \frac{14n_p^{s-1} + 4\hat{n}_p^{s-1}n_p^{s-1} - 2(n_p^{s-1})^2}{P-p+1},$$

then

$$\begin{aligned} & 14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 + \mathcal{A}(T^s_{\phi^{s-1}_{p+1}}) \\ & \geq \frac{14n_p^{s-1} + 4\hat{n}_p^{s-1}n_p^{s-1} - 2(n_p^{s-1})^2 + \mathcal{A}(T^s_{\phi^s_{P+1}-1})}{P-p+1} \end{aligned} \tag{5.9}$$

satisfies Eq. 5.4. From Eq. 5.5, if $\phi_{p+1}^s \neq \tau_r^s + 1$, *then*

$$\begin{aligned} & 14(\tilde{\phi}^{s-1} + w\rho) + 4\hat{n}_p^{s-1}(\tilde{\phi}^{s-1} + w\rho) - 2(\tilde{\phi}^{s-1} + w\rho)^2 \\ & = 14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 + \mathcal{A}(T^s_{\phi^{s-1}_{p+1}}) \end{aligned}$$

or if $\phi_{p+1}^s = \tau_r^s + 1$, *then*

$$\begin{aligned} & 14(\tilde{\phi}^{s-1} + 1) + 4\hat{n}_p^{s-1}(\tilde{\phi}^{s-1} + 1) - 2(\tilde{\phi}^{s-1} + 1)^2 \\ & = 14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 + \mathcal{A}(T^s_{\phi^{s-1}_{p+1}}). \end{aligned}$$

If $\tilde{\phi}^{s-1} + w\rho$ *for* $\tilde{\phi}^s$ *is substituted in the LB algorithm, then*

$$\begin{aligned} \phi_{p+1}^s &= \lceil \lceil \tilde{\phi}^{s-1} + w\rho \rceil / (w\rho) \rceil + \phi_p^s \\ &= \lceil \lceil \tilde{\phi}^{s-1} \rceil / (w\rho) \rceil + \phi_p^s + 1 \\ &= \phi_{p+1}^{s-1} + 1 \end{aligned}$$

and

$$\lceil \lceil \tilde{\phi}^{s-1} + 1 \rceil / (w\rho) \rceil + \phi_p^s \leq \lceil \lceil \tilde{\phi}^{s-1} + w\rho \rceil / (w\rho) \rceil + \phi_p^s.$$

Thus, from Eq. 5.9, $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.

PROPERTY 2.3 *If* $\phi_p^s = \phi_p^{s-1} + 1$ *and* $\tau_r^s = \tau_r^{s-1} + 1$, *then* $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.
Proof: *If*

$$\phi_{p+1}^s - \phi_{p+1}^{s-1} > 1.$$

and

$$14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 = \frac{14n_p^{s-1} + 4\hat{n}_p^{s-1}n_p^{s-1} - 2(n_p^{s-1})^2}{P-p+1}, \quad (5.10)$$

then

$$14\tilde{\phi}^{s-1} + 4(\hat{n}_p^{s-1} - w\rho)\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 < \frac{14n_p^{s-1} + 4(\hat{n}_p^{s-1} - w\rho)n_p^{s-1} - 2(n_p^{s-1})^2}{P-p+1}.$$

This implies that

$$\tilde{\phi}^{s-1} > \frac{n_p^{s-1}}{P-p+1}.$$

Substituting

$$\tilde{\phi}^{s-1} = \frac{n_p^{s-1}}{P-p+1} + \epsilon.$$

into Eq. 5.10 leads to a contradiction for $\epsilon > 0$. *Thus,* $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.

PROPERTY 2.4 *If* $\phi_p^s = \phi_p^{s-1} + 1$ *and* $\tau_r^s = \tau_r^{s-1}$ *then* $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.
Proof: *If*

$$\phi_{p+1}^s - \phi_{p+1}^{s-1} > 1.$$

and

$$14\tilde{\phi}^{s-1} + 4\hat{n}_p^{s-1}\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 = \frac{14n_p^{s-1} + 4\hat{n}_p^{s-1}n_p^{s-1} - 2(n_p^{s-1})^2}{P-p+1}, \quad (5.11)$$

then

$$14\tilde{\phi}^{s-1} + 4(\hat{n}_p^{s-1} - w\rho)\tilde{\phi}^{s-1} - 2(\tilde{\phi}^{s-1})^2 < \frac{14(n_p^{s-1} - w\rho) + 4(\hat{n}_p^{s-1} - w\rho)(n_p^{s-1} - w\rho) - 2(n_p^{s-1} - w\rho)^2}{P-p+1}.$$

This implies that

$$\tilde{\phi}^{s-1} > \frac{\frac{7}{2} + \hat{n}_p^{s-1} + \frac{w\rho}{2}}{P-p+1} > \frac{n_p^{s-1}}{P-p+1}.$$

From Property 2.3, this leads to a contradiction. Thus, $\phi_{p+1}^s - \phi_{p+1}^{s-1} \leq 1$.

For asynchronous message passing, the communication strategy is defined by the following procedure to manage the asynchronous send and receive operations.

Procedure: AP (Asynchronous Message Passing)

Step 1: If $1 < r$ then compute

$$\gamma_r^s = \min(m - \tilde{s} + (\phi_r^s - 2)(h\psi + w\rho) \\ -(r-2)(w\rho + 1)\min(0, \phi_r^s - \phi_{r-1}^s - 1) + w\rho, m + 1);$$

else set $\gamma_1^s = 1$

Step 2: If $r < P$ then compute

$$\gamma_{r+1}^s = \min(m - \tilde{s} + (\phi_{r+1}^s - 2)(h\psi + w\rho) \\ -(r-1)(w\rho + 1)\min(0, \phi_{r+1}^s - \phi_r^s - 1) + w\rho, m + 1);$$

else set $\gamma_{P+1}^s = m + 1$

Step 3: If $s = 1$ then go to Step 13; else continue

Step 4: If $r - 1$ is ODD go to Step 9; else continue

Step 5: If $r \neq 1$ and $\gamma_r^{s-1} < \gamma_r^s$ then send rows $[\gamma_r^{s-1} : \gamma_r^s - 1]$ to $r - 1$

Step 6: If $r \neq P$ and $\gamma_{r+1}^{s-1} > \gamma_{r+1}^s$ then send rows $[\gamma_{r+1}^s : \gamma_{r+1}^{s-1} - 1]$ to $r + 1$

Step 7: If $r \neq 1$ and $\gamma_r^s < \gamma_r^{s-1}$ then receive rows $[\gamma_r^s : \gamma_r^{s-1} - 1]$ from $r - 1$

Step 8: If $r \neq P$ and $\gamma_{r+1}^s > \gamma_{r+1}^{s-1}$ then receive rows $[\gamma_{r+1}^{s-1} : \gamma_{r+1}^s - 1]$ from $r + 1$

Step 9: Stop

Step 10: If $r \neq P$ and $\gamma_{r+1}^s > \gamma_{r+1}^{s-1}$ then receive rows $[\gamma_{r+1}^{s-1} : \gamma_{r+1}^s - 1]$ from $r + 1$

Step 11: If $r \neq 1$ and $\gamma_r^s < \gamma_r^{s-1}$ then receive rows $[\gamma_r^s : \gamma_r^{s-1} - 1]$ from $r - 1$

Step 12: If $r \neq P$ and $\gamma_{r+1}^{s-1} > \gamma_{r+1}^s$ then send rows $[\gamma_{r+1}^s : \gamma_{r+1}^{s-1} - 1]$ to $r + 1$

Step 13: If $r \neq 1$ and $\gamma_r^{s-1} < \gamma_r^s$ then send rows $[\gamma_r^{s-1} : \gamma_r^s - 1]$ to $r - 1$

Step 14: Stop

A new asynchronous message passing version of the PFG algorithm is presented below. It includes one additional input parameter p. The parameter $p \in \{1, 2, \ldots, P\}$ is a unique integer identifying the processor.

Algorithm: ASYNC (Asynchronous Version of the PFG)

Input$(A, D, P, w, h, d, \psi, \rho, p)$

$[m, n]$ =dimensions(A)

Compute S from Eq. 5.3

For $s = 1$ to S

Compute Φ_s using the LB Algorithm

Compute Γ_s using Eq. 5.8

For $\tau = \phi_p^s + 1$ to $\phi_{p+1}^s - 2$

Apply rotations in task T_τ^s

End For

Communicate using the AP Procedure

If $\phi_p^s < \phi_{p+1}^s$ then apply rotations in task $T_{\phi_p^s}^s$

If $\phi_p^s < \phi_{p+1}^s - 1$ then apply rotations in task $T_{\phi_{p+1}^s - 1}^s$

End For

Output(A, D)

Although not explicitly shown here, separating the receive and send operations in time is often recommended and can improve performance depending upon the implementation of the asynchronous operations.

2.3 Hybrid Shared Memory/Message Passing

The hybrid version of the parallel fast Givens algorithm derives the majority of its structure from the message passing version described in the previous section. For each processor $r \in \{1, 2, \ldots, P_m\}$ in the message passing environment, the hybrid version employs shared memory directives to distribute its work to S shared memory processors. From an organizational perspective, message passing plays the supervisory role. However, from a computational perspective, the hybrid version makes no distinction between the environments.

The hybrid and message passing versions differ in how the communication requirements for each message passing processor r are distributed to separate shared memory processors in the hybrid version where it is assumed that $S > 1$. The hybrid version employs the LB algorithm to compute a partition Φ_s for $P = P_m P_s$. Each processor $r \in \{1, 2, \ldots, P_m\}$ is assigned P_s independent groups of tasks. A shared memory directive distributes these groups of tasks to S processors for concurrent execution. While the communication network

is also a linear array, the basic building block is no longer a single processor. For the hybrid, a linear array constructed around clusters of processors that share memory is assumed. The burden of communicating with the neighbors $r-1$ and $r+1$ can be distributed to separate processors $p \in \{p_l, p_l+1, \ldots, p_r\}$ in the shared memory environment where $p_l = (r-1)P_s$ and $p_r = rP_s r - 1$ are the left and right shared memory processors for communication purposes. The following procedure is used to manage communication in the hybrid:

Procedure: HMP (Hybrid Message Passing)

Step 1: Set $p_l = (r-1)P_s$ and $p_r = rP_s - 1$

Step 2: Compute Γ_s from Eq. 5.8

Step 3: If $s = 1$ then go to Step 15; else continue

Step 4: If $r-1$ is ODD go to Step 10; else continue

Step 5: If $p = p_l$ and $\gamma_r^{s-1} < \gamma_r^s$ then send rows $[\gamma_r^{s-1} : \gamma_r^s - 1]$ to $r-1$

Step 6: If $p = p_r$ and $\gamma_{r+1}^{s-1} > \gamma_{r+1}^s$ then send rows $[\gamma_{r+1}^s : \gamma_{r+1}^{s-1} - 1]$ to $r+1$

Step 7: If $p = p_l$ and $\gamma_r^s < \gamma_r^{s-1}$ then receive rows $[\gamma_r^s : \gamma_r^{s-1} - 1]$ from $r-1$

Step 8: If $p = p_r$ and $\gamma_{r+1}^s > \gamma_{r+1}^{s-1}$ then receive rows $[\gamma_{r+1}^{s-1} : \gamma_{r+1}^s - 1]$ from $r+1$

Step 9: Stop

Step 10: If $p = p_r$ and $\gamma_{r+1}^s > \gamma_{r+1}^{s-1}$ then receive rows $[\gamma_{r+1}^{s-1} : \gamma_{r+1}^s - 1]$ from $r+1$

Step 11: If $p = p_l$ and $\gamma_r^s < \gamma_r^{s-1}$ then receive rows $[\gamma_r^s : \gamma_r^{s-1} - 1]$ from $r-1$

Step 12: If $p = p_r$ and $\gamma_{r+1}^{s-1} > \gamma_{r+1}^s$ then send rows $[\gamma_{r+1}^s : \gamma_{r+1}^{s-1} - 1]$ to $r+1$

Step 13: If $p = p_l$ and $\gamma_r^{s-1} < \gamma_r^s$ then send rows $[\gamma_r^{s-1} : \gamma_r^s - 1]$ to $r-1$

Step 14: Stop

Note, if $P_s = 1$ and $P_m > 1$, then the burden of communicating with the neighboring processors is managed sequentially by processor r. The hybrid version is described as follows:

Algorithm: Hybrid Version of the PFG

Input$(A, D, P_m, P_s, w, h, d, \psi, \rho, r)$

$[m, n]$ =dimensions(A)

Compute S from Eq. 5.3 and set $p_l = (r-1)P_s$ and $p_r = rP_s - 1$

For $s = 1$ to S

Compute Φ_s using the LB Algorithm for $P = P_m P_s$

PARALLEL LOOP

For $p = p_l$ to p_r

If $p = p_l$ Then

Communicate using the HMP Procedure

End If

If $p = p_r$ Then

Communicate using the HMP Procedure

End If

Apply rotations in tasks $[T^s_{\phi^s_p}, T^s_{\phi^s_p+1}, \ldots, T^s_{\phi^s_{p+1}-1}]$

End For

End For

Output(A, D)

3. Related Work

Parallel orthogonalization algorithms based on Givens rotations have been widely studied. During the past 5 years, however, the widespread availability of tuned kernels for performing matrix-matrix multiplication has relegated Givens-based algorithms to specialized applications only. Unlike Householder-based solution procedures, the component computations of Givens-based solution procedures cannot be efficiently cast in terms of matrix-matrix multiplication. As a consequence, most of research on parallel Givens-based algorithms predates 1995, including the following:

Sameh and Kuck, 1978

Lord et al., 1983

Modi and Clarke, 1984

Cosnard et al., 1986

Dongarra et al., 1986

Chamberlain and Powell, 1988

Porta, 1988

Chu and George, 1989

Jainandunsing and Deprettere, 1989

Pothen and Raghavan, 1989

Louka and Tchuente, 1988

Wright, 1991

Larriba-Pey et al., 1992

Badia et al., 1994

Meyer and Pascale, 1995

Wilburn et al., 1996

Lucka et al., 1996

Maslennikow et al., 1998

Many of the proposed methods are based on a regularized mapping of the matrix elements to processors. For instance, Lord et al. map the matrix by columns and by diagonal bands. Pothen and Raghavan map the matrix by row, and the rows are wrapped to a ring of processors. A distinguishing feature of the approach used in this chapter is the atypical mapping of the matrix elements to processors. The mapping scheme distributes the computational work of a group of independent tasks as evenly as possible.

4. Experimental Results

The experimental results are divided across two sections: Sections 4.1 and 4.2. Section 4.1 contrasts and compares the three implementations of the parallel fast Givens PFG (shared memory), SYNC (synchronous message passing), and ASYNC (asynchronous message passing). The results compare the minimum execution times of the implementations for various values of m and n on a 128-processor SGI Origin 2000 and 64-processor HP SPP-2000. Section 4.2 examines the performance characteristics of a hybrid shared memory/message passing version of the PFG algorithm on a 32-processor IBM SP3 as well as the HP and the SGI.

4.1 Shared Memory and Message Passing

For comparison purposes with PFG, SYNC, and ASYNC, the results in this section include the execution times of three competing parallel algorithms. These are two shared memory versions of LAPACK's DGEQRF: SGIMATH-DGEQRF and MLIB-DGEQRF available with the mathematical subroutine libraries MLIB and SGIMATH for the HP and SGI respectively, and ScaLAPACK's distributed QR factorization algorithm ScaLAPACK-PDGEQRF. All three algorithms require a user-specified parameter LWORK, and the recommended value returned in the output parameter WORK is used. For ScaLAPACK-PDGEQRF, the values of the four user-defined parameters are determined experimentally. The parameters are the number of P_r rows and P_c columns in the process grid, and blocking factor variables b_r and b_c for controlling block cyclic distribution.

Before SYNC and ASYNC can be executed, the matrix A must be stored in the local memory of processor 1. Upon completion, the upper triangular result also is stored in processor 1. To compare the performance characteristics of ScaLAPACK's distributed QR factorization algorithm PDGEQRF with our message passing implementations, the reported execution times for ScaLAPACK-PDGEQRF include the cost of distributing the matrix A from processor 1 to the other $P-1$ processors. This includes the time to broadcast the matrix A from processor 1 to the other processors and the time to pack the data on the local processor for block cyclic execution. Even though the upper triangular result is also distributed across P processors in packed format upon completion, this is not included in the costs associated with gathering the final result and storing it on processor 1. The true nature of this cost would depend on the processing to follow.

The number of integer combinations of the parameters that satisfy $1 \leq h\psi \leq m$, $1 \leq w\rho \leq n$, and $1 \leq d\rho \leq n$ is potentially large and devising an efficient procedure for choosing optimal or near optimal combinations is beyond the scope of this book. Nonetheless, some definitive patterns have emerged in the results that can guide the end user in selecting the combinations that produce high levels of performance. These patterns are discussed in the following paragraphs.

Tables 5.1 and 5.3 present the minimum execution times of the three implementations of the parallel fast Givens algorithms PFG, SYNC, and ASYNC; two vendor-tuned, shared memory versions of LAPACK's DGEQRF algorithm MLIB-DGEQRF and SGIMATH-DGEQRF; and a version of ScaLAPACK's distributed QR factorization algorithm PDGEQRF for the SGI ScaLAPACK-PDGEQRF. The three implementations outperform the competing algorithms for all values of m and n on both the SGI and the HP. For the cases where $m \geq n = 1500$ and $500 \geq m \geq n = 100$, the lowest execution times are obtained with the PFG algorithm on the SGI. The lowest execution

HP Algorithm Implementation	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
PFG Shared Memory	2.65	1.11	0.240	0.088	0.0198
SYNC Blocking MPI	2.00	0.95	0.293	0.124	0.0177
ASYNC Non-Blocking MPI	1.98	0.86	0.287	0.106	0.0190
MLIB Shared Memory	3.90	1.19	0.570	0.212	0.0426

Table 5.1. Minimum execution times in seconds on the HP SPP-2000.

times for $1500 \geq m \geq n = 500$ are obtained with the PFG algorithm on the HP. The corresponding parameter settings for the SGI and HP are presented in Tables 5.2 and 5.4, respectively.

HP Parameter		$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
P	PFG	16	16	16	16	7
	SYNC	48	42	24	18	10
	ASYNC	45	50	18	12	7
w	PFG	3	2	2	5	4
	SYNC	2	4	2	2	3
	ASYNC	2	2	2	2	2
h	PFG	3	3	8	7	12
	SYNC	6	6	4	4	10
	ASYNC	6	7	6	7	9
d	PFG	500	500	167	167	34
	SYNC	500	500	167	167	34
	ASYNC	500	500	167	167	34
ψ	PFG	2	2	2	2	2
	SYNC	2	2	2	2	2
	ASYNC	2	2	2	2	2
ρ	PFG	3	3	3	3	3
	SYNC	3	3	3	3	3
	ASYNC	3	3	3	3	3

Table 5.2. Optimal parameter settings on the HP SPP-2000 for Table 5.1.

In Sections 1.1 and 1.2, superscalar parameters ψ and ρ and the cache parameters h and d are introduced, respectively. The parameters control the

SGI Algorithm / Implementation	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
PFG Shared Memory	1.74	1.01	0.283	0.114	0.0153
SYNC Blocking MPI	2.38	1.14	0.323	0.111	0.0180
ASYNC Non-Blocking MPI	2.41	1.08	0.281	0.111	0.0168
SGIMATH Shared Memory	10.00	3.32	1.230	0.294	0.0440
ScaLAPACK BLACs	4.11	2.89	0.728	0.260	0.0420

Table 5.3. Minimum execution times in seconds on the SGI Origin 2000.

amount of reuse in the register bank and the caches. Along with the load balancing parameter w introduced in Section 1.3, the parameters h, ψ, and ρ also control the number of rotations that comprise a task or the relative amount of fine-grain and coarse-grain parallelism present in the algorithm. Despite the fact that no quantitative model to characterize this tradeoff exists, handful of qualitative observations can be gleaned from the results to guide the end-user in choosing a parameter combination that results in high levels of performance. From Tables 5.2 and 5.4, it is clear that the superscalar parameters are invariant to the machine, implementation, and problem dimensions. For $h > 1$, the parameter d controls reuse in the L-1 cache between two adjoining groups of $\psi\rho$ rotations. As a result, optimal settings for d are largely invariant to m, n, P, w, and h. Variations in the settings for the SGI can be attributed to the fact that the L-1 cache is large enough to store $(2\psi + \rho)n$ elements for $n \leq 500$. As a consequence, the parameterization is unnecessary and is effectively disabled by setting $d \geq \lceil 500/\rho \rceil$. Because the HP has only a single cache, the parameterization is disabled for all matrix dimensions. Making the number of rotations that comprise a task as small as possible reduces load imbalance. For fixed ψ and ρ, this means reducing h and w. Reuse in the L-2 cache, however, improves with increasing h and to a lesser extent with increasing $w > 1$ until the capacity of the cache is reached. Because optimal settings for h and w are nearly constant for various values of m and n, the benefits of reducing load imbalance outweigh the benefits of reuse. For both the SGI and the HP, optimal settings for w fall between 2 and 5. Optimal settings for h fall between 5 and 16 on the SGI and between 3 and 12 on the HP. The cardinality of the largest concurrency provides the

following upper bound on P:

$$P \leq \left\lceil \frac{h\psi \left\lceil \frac{m-1}{h\psi} - 1 \right\rceil + h\psi - 1}{h\psi + w\rho} \right\rceil .$$

For instance, $P \leq 100$ for the case S1 in Table 5.4.

SGI Parameter		$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
P	PFG	96	91	38	16	8
	SYNC	32	25	15	19	7
	ASYNC	28	37	13	13	7
w	PFG	2	2	2	5	4
	SYNC	2	2	2	2	2
	ASYNC	2	2	2	2	2
h	PFG	12	5	16	7	7
	SYNC	5	7	7	7	7
	ASYNC	7	5	7	6	7
d	PFG	176	176	167	167	34
	SYNC	176	176	167	167	34
	ASYNC	176	176	167	167	34
ψ	PFG	2	2	2	2	2
	SYNC	2	2	2	2	2
	ASYNC	2	2	2	2	2
ρ	PFG	3	3	3	3	3
	SYNC	3	3	3	3	3
	ASYNC	3	3	3	3	3

Table 5.4. Optimal parameter settings on the SGI Origin 2000 for Table 5.3.

In Figures 5.7 and 5.8, the sensitivity of the execution times to P for experimentally determined optimal values of the parameters w, h, d, ψ, and ρ and for the case $m = 3000$ and $n = 1500$ are explored. For comparison purposes, th optimal execution times of ScaLAPACK-PDGEQRF in Figure 5.8 and MLIB-DGEQRF are included in Figure 5.7. SYNC and ASYNC consistently outperform MLIB-DGEQRF and ScaLAPACK-PDGEQRF. In addition, Figure 5.8 shows that the shared memory implementation outperforms the two message passing implementations on the SGI. On the HP, both message passing implementations outperform the shared memory implementation as depicted In Figure 5.7. SGIMATH-DGEQRF is omitted in Figure 5.8 because all execution times were above 6.0 seconds. For the HP, the shared memory algorithm achieves its best results when $P \leq 16$ or when the job is limited to a single hypernode. Performance degrades for each additional hypernode added at $P = 17$, $P = 33$, and $P = 49$. The benefits of $P > 16$ are

negated entirely by the cost of running a shared memory job across multiple hypernodes.

Tables 5.2 and 5.4 show the optimal parameter settings for the execution times presented in Table 5.1. Tables 5.5 and 5.6 compare the optimal execution times with those execution times obtained using the optimal parameter settings for the SGI on the HP and for the HP on the SGI. In most cases, using suboptimal parameters leads to dramatically slower results. This is most evident on the SGI where suboptimal parameter settings in some cases nearly double the execution time. Overall, the HP is less sensitive to changes in the parameters. Execution times could not be obtained for cases S1 and S4 on the HP because the cases exceeded the number of processors available on the system.

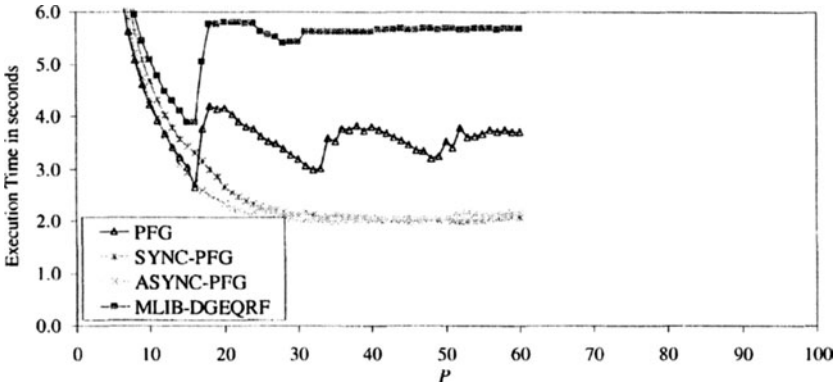


Figure 5.7. Minimum execution times as function of the number of processors P on the HP for the case $m = 3000$ and $n = 1500$.

4.2 Hybrid Message Passing/Shared Memory

To investigate whether or not blending message passing and shared memory programming environments enhance performance, extensive timing experiments were conducted on a 64-processor HP SPP-2000, a 128-processor SGI Origin 2000, and a 32-processor IBM SP3 using the hybrid version of the parallel fast Givens algorithm. Presented in this section is a small subset of the collected data and a summary of our findings.

Our evaluation compares the same four problems on all three machines. The reported execution times are based on the following initial and final conditions: 1) before execution can begin, the matrix A must be stored in the local memory of processor 1; and 2) upon completion, the upper triangular result is stored in the local memory of processor 1. In addition, the reported

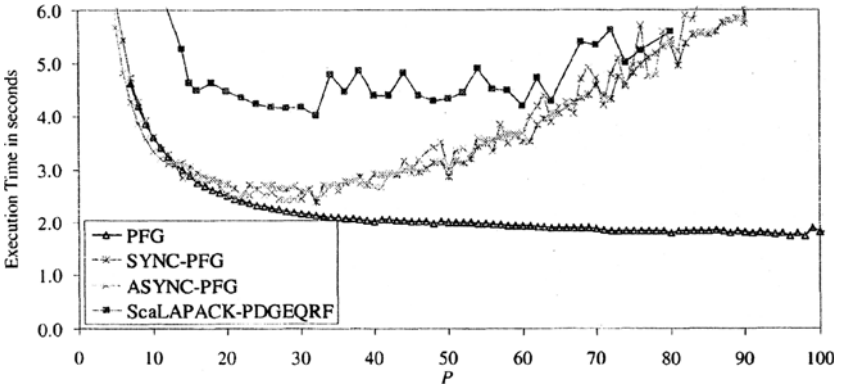


Figure 5.8. Minimum execution times as function of the number of processors P on the SGI for the case $m = 3000$ and $n = 1500$.

HP Algorithm / Implementation	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
PFG	2.65	1.11	0.240	0.088	0.0198
Shared Memory	NA	NA	0.313	0.088	0.0337
SYNC	2.00	0.95	0.293	0.124	0.0177
Blocking MPI	2.40	1.22	0.355	0.127	0.0235
ASYNC	1.98	0.86	0.287	0.106	0.0190
Non-blocking MPI	2.25	1.03	0.320	0.120	0.0229

Table 5.5. Sensitivity to variations in the parameter settings on the HP SPP-2000.

execution times are based on experimentally determined optimal values of the blocking parameters. Performance in Mflops is based on the assumption that the number of floating-point operations to compute the QR factorization of a real $m \times n$ matrix is approximately $2n^2(m - n/3)$.

The HP SPP-2000 is comprised of four shared memory nodes with 16 processors. The IBM SP3 also is comprised of four shared memory nodes with each node containing eight processors. The SGI is comprised of 64 nodes with each node containing two processors.

As evidenced in Tables 5.7-5.10, mixed blends – $P_m > 1$ and $P_s > 1$ – enhance algorithm performance on the HP and IBM for $P > 16$. In particular, when P is divisible by two, mixed blends outperform all others on the HP.

SGI Algorithm / Implementation	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
PFG Shared Memory	1.74 3.55	1.01 1.79	0.283 0.380	0.114 0.114	0.0153 0.0155
SYNC Blocking MPI	2.38 3.10	1.14 1.37	0.323 0.365	0.111 0.139	0.0180 0.0187
ASYNC Non-blocking MPI	2.41 5.72	1.08 1.92	0.281 0.657	0.111 0.274	0.0168 0.0179

Table 5.6. Sensitivity to variations in the parameter settings on the SGI Origin 2000.

Architectural constraints limit shared memory and message passing operation to eight and 16 processors, respectively on the IBM. As a consequence, blends for $P \in \{17, 19, 22, 23, 25, 27, 29, 30, 31\}$ are not available (NA) on the system. For $P \in \{18, 20, 21, 24, 28, 32\}$, only mixed blends are possible. As a general rule, minimizing the number of message passing processors delivers the best performance on the IBM. For the SGI, shared memory outperforms all other blends in almost every case regardless of P.

While Tables 5.7-5.10 show blending to be beneficial, the magnitude of the benefits is unclear. In Tables 5.11, 5.12, and 5.13, execution times for all possible blends in the case $P = 6$, $P = 12$, and $P = 24$ for each of three machines are presented. These special cases are highlighted because they have at least four blends, and as many as eight. In addition, the HP SPP-2000, IBM SP3, and SGI Origin 2000 are constructed around nodes of 16, eight, and two processors, respectively. These cases were chosen to encompass multiple nodes for the machines. For example, in Table 5.11 the best and worst blends differ by an astounding 371 seconds on the HP for $P = 24$, $m = 3000$, and $n = 1500$. While differences on the IBM and SGI are not nearly as significant, the best blends still outperform the worst blends by a wide margin in many cases. In general, where M exceeds the number of nodes on the machine, performance suffers. While the number of nodes on the SGI is large, the HP and IBM are comprised of only four nodes.

The benefits of hybrid parallelism in the context of QR factorization are clear. By manipulating the number of shared memory processors and message passing processors, high levels of performance can be extracted from three machines on a variety of problems. Neither pure shared memory, pure message passing, mixed blends, nor any particular computer architecture delivers consistently superior performance for $P = 1, 2, \ldots, 32$ and various values of

	HP			IBM			SGI		
P	M	S	Time (s)	M	S	Time (s)	M	S	Time (s)
1	1	1	32.51	1	1	18.03	1	1	22.80
2	1	2	16.97	1	2	9.91	1	2	13.54
3	1	3	11.94	1	3	6.95	1	3	9.07
4	1	4	9.12	1	4	5.55	1	4	7.23
5	1	5	7.43	1	5	4.70	1	5	6.09
6	1	6	6.26	1	6	4.09	1	6	5.34
7	1	7	5.43	1	7	3.80	1	7	4.85
8	1	8	4.84	1	8	3.49	1	8	4.36
9	1	9	4.35	1	9	3.17	1	9	4.15
10	1	10	4.02	1	10	2.97	1	10	3.82
11	1	11	3.72	1	11	2.81	1	11	3.60
12	1	12	3.41	1	12	2.68	1	12	3.45
13	1	13	3.20	1	13	2.62	1	13	3.32
14	1	14	3.00	1	14	2.52	1	14	3.17
15	1	15	2.81	1	15	2.44	1	15	3.10
16	1	16	2.69	8	2	2.34	1	16	2.98
17	17	1	2.82		NA		1	17	2.92
18	2	9	2.50	6	3	2.49	1	18	2.84
19	19	1	2.64		NA		1	19	2.75
20	2	10	2.35	5	4	2.38	1	20	2.70
21	21	1	2.53	7	3	2.29	1	21	2.63
22	2	11	2.20		NA		1	22	2.64
23	23	1	2.38		NA		1	23	2.59
24	2	12	2.07	8	3	2.12	1	24	2.54
25	25	1	2.27		NA		1	25	2.50
26	2	13	2.00		NA		1	26	2.47
27	27	1	2.19		NA		1	27	2.44
28	2	14	1.91	7	4	2.15	1	28	2.38
29	29	1	2.17		NA		1	29	2.36
30	2	15	1.86		NA		1	30	2.33
31	31	1	2.14		NA		1	31	2.33
32	2	16	1.80	8	4	1.92	1	32	2.34

Table 5.7. Execution times as a function of P for experimentally determined optimal blends on the HP SPP-2000, IBM SP3, and SGI Origin 2000 in the case $m = 3000$ and $n = 1500$.

m and n. At least in the context parallel QR factorization, generalizations regarding the superiority of one programming environment over another are clearly misleading. Though for $P \leq 16$, pure message passing on the IBM generally outperforms all other blends on the HP, IBM, and SGI. For $P > 16$, the ranking of machines and programming environments is highly dependent on the value of P. The hybrid version affords the end user the necessary flexibility to navigate this complex operating environment.

	HP			IBM			SGI		
P	M	S	Time (s)	M	S	Time (s)	M	S	Time (s)
1	1	1	12.91	1	1	7.23	1	1	9.05
2	1	2	6.44	2	1	4.01	1	2	5.37
3	1	3	4.48	3	1	2.92	1	3	3.84
4	1	4	3.66	4	1	2.34	1	4	3.14
5	1	5	2.96	5	1	2.06	1	5	2.74
6	1	6	2.54	6	1	1.85	1	6	2.45
7	1	7	2.23	7	1	1.68	1	7	2.23
8	1	8	1.99	8	1	1.57	1	8	2.05
9	1	9	1.80	9	1	1.51	1	9	1.98
10	1	10	1.65	10	1	1.44	1	10	1.90
11	1	11	1.52	11	1	1.37	1	11	1.81
12	1	12	1.42	12	1	1.33	1	12	1.75
13	1	13	1.33	13	1	1.30	1	13	1.70
14	1	14	1.26	2	7	1.22	1	14	1.63
15	1	15	1.20	15	1	1.21	1	15	1.55
16	1	16	1.16	2	8	1.06	1	16	1.51
17	17	1	1.21		NA		1	17	1.49
18	2	9	1.18	3	6	1.18	1	18	1.46
19	19	1	1.27		NA		1	19	1.43
20	2	10	1.13	4	5	1.19	1	20	1.39
21	21	1	1.23	3	7	1.10	1	21	1.37
22	2	11	1.07		NA		1	22	1.37
23	23	1	1.18		NA		1	23	1.37
24	2	12	1.04	3	8	1.00	1	24	1.36
25	25	1	1.15		NA		1	25	1.36
26	2	13	0.98		NA		1	26	1.28
27	27	1	1.11		NA		1	27	1.33
28	2	14	0.97	4	7	0.99	1	28	1.34
29	29	1	1.10		NA		1	29	1.30
30	2	15	0.93		NA		1	30	1.29
31	31	1	1.10		NA		1	31	1.29
32	2	16	0.90	4	8	0.90	2	16	1.31

Table 5.8. Execution times as a function of P for experimentally determined optimal blends on the HP SPP-2000, IBM SP3, and SGI Origin 2000 in the case $m = 1500$ and $n = 1500$.

In conclusion, high levels of performance can be extracted from three machines in a variety of parallel programming environments and on a variety of problems. The algorithm outperforms all competing parallel QR factorization algorithms installed on the SGI Origin 2000 and on the HP SPP-2000. None of the machines delivers consistently superior performance for all matrix dimensions considered in this book. No single parallel programming environment emerges as a clear choice for high performance on the SGI or the HP.

	HP			IBM			SGI		
P	M	S	Time (s)	M	S	Time (s)	M	S	Time (s)
1	1	1	2.05	1	1	1.15	1	1	1.49
2	2	1	1.15	2	1	0.65	1	2	0.89
3	1	3	0.80	3	1	0.48	1	3	0.67
4	1	4	0.61	4	1	0.40	1	4	0.57
5	1	5	0.51	5	1	0.36	1	5	0.52
6	1	6	0.44	6	1	0.34	6	1	0.46
7	1	7	0.40	7	1	0.31	1	7	0.44
8	1	8	0.36	8	1	0.29	1	8	0.41
9	1	9	0.33	9	1	0.30	1	9	0.41
10	1	10	0.32	10	1	0.29	1	10	0.40
11	1	11	0.31	11	1	0.28	1	11	0.40
12	1	12	0.29	12	1	0.27	1	12	0.37
13	1	13	0.28	13	1	0.28	1	13	0.37
14	1	14	0.26	2	7	0.28	1	14	0.35
15	1	15	0.26	15	1	0.27	1	15	0.34
16	1	16	0.26	2	8	0.26	1	16	0.35
17	17	1	0.27		NA		1	17	0.35
18	2	9	0.27	3	6	0.28	1	18	0.33
19	19	1	0.32		NA		1	19	0.33
20	2	10	0.26	4	5	0.28	1	20	0.33
21	21	1	0.33	3	7	0.27	1	21	0.34
22	2	11	0.26		NA		1	22	0.32
23	23	1	0.34		NA		1	23	0.32
24	2	12	0.25	3	8	0.26	1	24	0.31
25	1	25	0.34		NA		1	25	0.31
26	2	13	0.24		NA		1	26	0.31
27	3	9	0.28		NA		1	27	0.32
28	2	14	0.23	4	7	0.29	1	28	0.32
29	1	29	0.31		NA		1	29	0.31
30	2	15	0.21		NA		1	30	0.32
31	1	31	0.29		NA		1	31	0.32
32	2	16	0.21	4	8	0.27	1	32	0.32

Table 5.9. Execution times as a function of P for experimentally determined optimal blends on the HP SPP-2000, IBM SP3, and SGI Origin 2000 in the case $m = 1500$ and $n = 500$.

The hybrid shared memory/message passing programming model shows the most promise and is necessary to fully utilize the IBM SP3.

	HP			IBM			SGI		
P	M	S	Time (s)	M	S	Time (s)	M	S	Time (s)
1	1	1	0.496	1	1	0.280	1	1	0.363
2	1	2	0.280	2	1	0.172	2	1	0.246
3	1	3	0.205	3	1	0.142	3	1	0.214
4	1	4	0.169	4	1	0.121	2	2	0.184
5	1	5	0.144	5	1	0.116	5	1	0.177
6	1	6	0.128	6	1	0.111	2	3	0.163
7	1	7	0.120	7	1	0.108	7	1	0.158
8	1	8	0.112	1	8	0.102	8	1	0.154
9	1	9	0.105	9	1	0.109	1	9	0.153
10	1	10	0.100	2	5	0.099	1	10	0.149
11	1	11	0.098	11	1	0.099	1	11	0.144
12	1	12	0.097	2	6	0.097	1	12	0.140
13	1	13	0.095	13	1	0.099	1	13	0.139
14	1	14	0.093	2	7	0.095	14	1	0.141
15	1	15	0.091	15	1	0.099	15	1	0.137
16	1	16	0.089	2	8	0.088	1	16	0.138
17	1	17	0.088		NA		1	17	0.134
18	2	9	0.093	3	6	0.098	1	18	0.132
19	1	19	0.104		NA		1	19	0.128
20	2	10	0.090	4	5	0.101	1	20	0.127
21	1	21	0.108	3	7	0.095	1	21	0.125
22	2	11	0.088		NA		1	22	0.129
23	23	1	0.107		NA		1	23	0.129
24	2	12	0.088	3	8	0.089	1	24	0.128
25	25	1	0.114		NA		1	25	0.127
26	2	13	0.086		NA		1	26	0.131
27	3	9	0.099		NA		1	27	0.132
28	2	14	0.087	4	7	0.096	1	28	0.127
29	1	29	0.116		NA		1	29	0.126
30	2	15	0.087		NA		1	30	0.128
31	1	31	0.116		NA		1	31	0.129
32	2	16	0.088	4	8	0.090	1	32	0.140

Table 5.10. Execution times as a function of P for experimentally determined optimal blends on the HP SPP-2000, IBM SP3, and SGI Origin 2000 in the case $m = 500$ and $n = 500$.

HP			$m = 3000$	$m = 1500$	$m = 1500$	$m = 500$
P	M	S	$n = 1500$	$n = 1500$	$n = 500$	$n = 500$
1	6		6.26	2.54	0.443	0.443
2	3		6.37	2.68	0.471	0.471
3	2		7.27	2.89	0.543	0.543
6	1		6.68	2.72	0.495	0.495
1	12		3.41	1.42	0.292	0.292
2	6		3.47	1.55	0.312	0.312
3	4		4.19	1.78	0.394	0.394
4	3		4.59	2.11	0.419	0.419
6	2		4.28	1.81	0.476	0.476
12	1		3.73	1.58	0.359	0.359
1	24		3.74	1.60	0.354	0.354
2	12		2.07	1.04	0.247	0.247
3	8		2.65	1.24	0.331	0.331
4	6		3.06	1.57	0.340	0.340
6	4		2.64	1.24	0.318	0.318
8	3		94.81	1.38	0.416	0.416
12	2		375.35	1.38	0.484	0.484
24	1		2.32	1.17	0.348	0.348

Table 5.11. Execution times in seconds for various blends on the HP SPP-2000.

IBM			$m = 3000$	$m = 1500$	$m = 1500$	$m = 500$
P	M	S	$n = 1500$	$n = 1500$	$n = 500$	$n = 500$
1	6		4.60	1.90	0.416	0.416
2	3		4.72	2.11	0.389	0.389
3	2		4.77	2.25	0.430	0.430
6	1		4.09	1.85	0.337	0.337
1	12			NA		
2	6		2.91	1.36	0.296	0.296
3	4		3.07	1.47	0.320	0.320
4	3		3.20	1.55	0.351	0.351
6	2		11.08	5.59	1.67	1.669
12	1		2.68	1.33	0.275	0.275
1	24			NA		
2	12			NA		
3	8		2.12	0.996	0.257	0.257
4	6		2.26	1.11	0.288	0.288
6	4		6.32	1.37	0.366	0.366
8	3		17.40	1.29	0.376	0.376
12	2		7.23	1.51	0.830	0.830
24	1			NA		

Table 5.12. Execution times in seconds for various blends on the IBM SP3.

SGI			$m = 3000$	$m = 1500$	$m = 1500$	$m = 500$
P	M	S	$n = 1500$	$n = 1500$	$n = 500$	$n = 500$
	1	6	5.34	2.45	0.476	0.476
	2	3	5.44	2.53	0.484	0.484
	3	2	13.18	5.87	0.678	0.678
	6	1	5.68	2.88	0.459	0.459
	1	12	3.45	1.75	0.374	0.374
	2	6	3.56	1.85	0.408	0.408
	3	4	9.05	3.97	0.575	0.575
	4	3	5.29	2.72	0.771	0.771
	6	2	5.93	3.20	0.996	0.996
	12	1	3.61	1.86	0.387	0.387
	1	24	2.54	1.36	0.314	0.314
	2	12	2.72	1.43	0.353	0.353
	3	8	6.64	3.38	1.410	1.413
	4	6	4.36	2.35	0.589	0.589
	6	4	4.86	2.78	0.985	0.985
	8	3	5.32	3.00	1.190	1.194
	12	2	4.06	2.09	0.902	0.902
	24	1	3.18	1.61	0.438	0.438

Table 5.13. Execution times in seconds for various blends on the SGI Origin 2000.

Chapter 6

CASE STUDY 2: PARALLEL COMPACT WY QR FACTORIZATION

During the past five years the widespread availability of tuned kernels for performing matrix-matrix multiplication has dramatically narrowed the focus of parallel algorithm research in the field of linear algebra. Underlying this change is the fact that an efficient subroutine can exploit properties of the processor superscalar design and memory hierarchies to compute a matrix-matrix multiplication faster than a subroutine can sequentially compute the component matrix-vector multiplications. Indeed, studies have shown that substantial gains in performance can be realized by redesigning linear algebra algorithms to increase the percentage of operations performed as matrix-matrix multiplication (Bischof et al., 1994; Dongarra et al., 1989; Gallivan et al., 1988; Schreiber and Van Loan, 1989). This is evidenced on the SGI POWER Challenge where LAPACK reports an efficiency of 268 Mflops when multiplying two 1000×1000 matrices, but only 41 Mflops when multiplying a 1000×1000 matrix and a 1000 element vector (Anderson et al., 1995). A potential six-fold increase in performance is strong impetus for developing algorithms whose computations can be expressed in terms of matrix-matrix multiplication instead of matrix-vector multiplication. Solution procedures whose component computations cannot be cast in terms of matrix-matrix multiplication are no longer the focus of much research.

Although parallel QR factorization is an important research topic , only a handful of parallel algorithms have been designed to employ matrix-matrix multiplication in the last 25 years. These algorithms partition the underlying matrix data into two-dimensional blocks and distribute these blocks in a cyclical fashion to P processors. Block computations proceed concurrently with little need for communication or synchronization, and the computations are dominated by matrix-matrix multiplication. Unfortunately the data distribution schemes associated with these algorithms, such as ScaLAPACK's

PDGEQRF (Blackford et al., 1997; Choi et al., 1995), are not always compatible with the data movement patterns of software applications. For high performance signal processing applications where strict latency constraints prohibit repartitioning and redistributing the data to accommodate a particular distribution pattern, the mismatch can degrade performance. The cost of repartitioning and redistributing the data is usually proportional to the number of processors and has the effect of hindering overall scalability. Where mismatches occur, these algorithms afford the end user little flexibility to meet hard latency constraints by manipulating the number of processors. Note that some very recent work on efficient real time redistribution techniques promises to make these algorithms more attractive to embedded applications (Park et al., 1999; Petit and Dongarra, 1999).

Anything short of incorporating real-time redistribution techniques into these algorithms, however, places an unacceptable burden on the algorithm designer. This is particularly true when the design process begins with a sequential specification. By profiling an implementation of the sequential specification on the target architecture, bottlenecks can be identified and evaluated as possible candidates for parallel execution. Parallel algorithms based on two-dimensional block cyclic distribution schemes hinder the ability of the algorithm designer to rapidly prototype solutions to these bottlenecks. Custom scatter/gather operations are necessary to integrate these algorithms into a sequential test bed. These operations may become new bottlenecks and may require extensive tuning. Ideally, designers want to deploy algorithms that closely match the data movement pattern of the existing application.

This chapter describes the design, implementation, and performance of a new parallel algorithm for computing the factorization $A = QR$ that is well suited to applications where block cyclic data distribution schemes degrade performance. The algorithm applies Householder reflections in block fashion to reduce a real $m \times n$ matrix A to upper triangular form where $m \geq n$. Using the "Compact WY" representation developed by Schreiber and Van Loan (1989), blocks of Householder reflections are aggregated so as to use matrix-matrix multiplication. User-defined parameters h, w, and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ control the aggregation of Householder reflections, the composition of two types of indivisible computational primitives or tasks, and the assignment of tasks to processors for concurrent execution. In contrast to existing parallel QR factorization algorithms that employ matrix-matrix multiplication, the multiprocessor partitioning strategy is not governed by an underlying static data distribution scheme. Tasks and their dependency relationships define a task dependency graph, and the parameters w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ partition the graph into $P < (n + 1/2)/h$ non-overlapping regions of tasks. Within a region, processors execute tasks and exchange messages asynchronously. The initial and final data resides on the same processor.

1. Parallel Compact WY Algorithm

This section presents the Parallel Compact WY (PCWY) algorithm. Superscalar and memory hierarchy parameterizations are discussed in Section 1.1. In Section 1.2, the multiprocessor parameters w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ are introduced.

1.1 Superscalar and Memory Hierarchy Parameterization

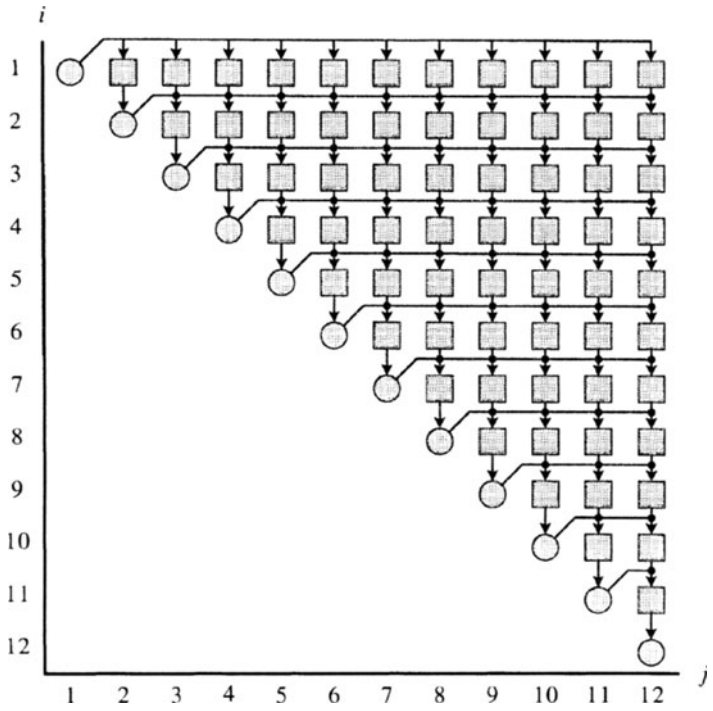


Figure 6.1. Task dependency graph for the case $h = 2$, $m \geq n$, and $n = 24$.

In the SCWY algorithm, the parameter h controls the aggregation of Householder reflections, and therefore the relative amount of computations performed as matrix-matrix multiplication. In contrast to the superscalar and memory hierarchy parameters presented in Chapter 5, the parameter h does not explicitly control reuse in the registers or the caches. However, tuned matrix-matrix multiplication kernels are generally register and cache efficient. However, kernel efficiency does depends on the dimensions of the matrices

and in particular, on the smallest dimension. For the SCWY algorithm, the smallest dimension is the parameter h.

For the PCWY algorithm, the matrix multiplication operations involved in applying the aggregated Householder reflections to A^k are divided into column segments of width h and denoted by the symbol $s_{i,j}$, hereafter referred to as a *subordinate task* where $j = i+1, i+2, \ldots, \tilde{n}$, $i = (k-1)/h + 1$, and $\tilde{n} = \lceil n/h \rceil$. Both the inner and outer-right dimensions of the matrix multiplication operations involved in applying the aggregated reflections are now equal to h. While this can have the effect of reducing kernel efficiency, the following subordinate tasks

$$s_{i,i+1}, s_{i,i+2}, \ldots, s_{i,\tilde{n}}$$

are independent and can be distributed to multiple processors for concurrent execution.

Before introducing the partitioning and sequencing strategy, the computations that comprise a *leading task* and the dependency relationships among tasks are described. Leading task l_i is comprised of the computations necessary to factor the submatrix $A^k_{k:m,k:\hat{n}}$ and determine Y^k and T^k where $\hat{n} = \min(k+h-1, n)$. From the underlying dependencies among computations, l_i depends on $s_{i-1,j}$, and $s_{i,j}$ depends on l_i and $s_{i-1,j}$. For the case $h = 2$ and $m \geq n$, and $n = 24$, the corresponding task dependency graph is shown in Figure 6.1 where each shaded circle represents a task l_i and each shaded square represents a task $s_{i,j}$.

1.2 Multiprocessor Parameterization

Using the task dependency graph as a geometric representation of the computational work involved in factoring the matrix A, the parameters w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ control a load balancing algorithm that partitions the graph into P diagonal bands of roughly equal area. The diagonal bands are bounded by $P-1$ diagonal lines with slope m_p, y-intercept b_p, and x-intercept $(x_p - 1)$ and P unity slope diagonal lines with y-intercept b'_p. The line equations are as follows:

$$\begin{aligned} y &= m_p x + b_p \\ y &= x + b'_p. \end{aligned}$$

The parameter $w > 1$ is the spacing in the x direction between the P unity slope diagonal lines. The parameters $\delta_1, \delta_2, \ldots, \delta_{P-1}$ are twiddle factors that adjust the relative amount of area contained in each band. The following algorithm computes the line parameters:

Algorithm: LB (Load Balancing)

Input(n, P, w, h)

$\tilde{n} = \lceil n/h \rceil$,

$a = (\tilde{n} + 1/2 - wP)^2/2$

$P = \min(P, \lfloor(\tilde{n} + w - 1/2)/w\rfloor)$

For $p = 1$ to $P - 1$

If $p = 1$ then $\alpha = \delta_1 a$

Else

$\alpha = r_p + (1 + \delta_p)/(P - p + 1)$

$a = (1 - (1 + \delta_p)/(P - p + 1))a$

End For

$x_p = (1 + \sqrt{1 + 8\alpha})/2$

$m_p = x_p$

$b_p = -m_p(x_p + pw - 1)$

$b'_p = -pw$

$r_p = x'_p(x_p - 1)/2$

$x_p = x_p + pw$

End For

$b'_P = -Pw$

If the Cartesian coordinates (x, y) of task l_i and $s_{i,j}$ are $(j, j - 1/2)$ and $(j, i - 1/2)$ respectively, then the following algorithm returns TRUE if task l_i or $s_{i,j}$ is assigned to processor p where $M = \{m_1, m_2, \ldots, m_{P-1}\}$, $B = \{b_1, b_2, \ldots, b_{P-1}\}$, and $B' = \{b'_1, b'_2, \ldots, b'_P\}$:

Algorithm: TP (Task Partitioning)

Input(x, y, p, M, B, B')

If $x < y$ then Output(TRUE)

Else If $y < 0$ then Output(TRUE)

Else If $p = 1$ and $\min(m_p x + b_p, x + b'_p) \leq y$ then Output(TRUE)

Else If $1 < p < NP$ and $\min(m_p x + b_p, x + b'_p) \leq y < \min(m_{p-1} x + b_{p-1}, x + b'_{p-1})$ then Output(TRUE)

Else If $y < \min(m_{p-1}x + b_{p-1}, x + b'_{p-1})$ then Output(TRUE)

Output(FALSE)

The sequencing strategy is straightforward. Processor 1 executes tasks first from top to bottom and then from left to right. Processors $2, 3, \ldots, P$ execute tasks first from left to right and then from top to bottom. For processor 1, the idea is to rapidly compute and distribute the aggregated reflections $(I + Y^k S^k Y^{kT})$ for $k = 1, h+1, 2h+1, \ldots, (\tilde{n}-1)h+1$. For the remaining processors, the idea is to execute tasks in such an order as to delay the need for each set of aggregated reflections as long as possible. Figures 6.2 and 6.3 depict the partitioning and sequencing strategy. The task numbering depicts not only the sequencing strategy, but also simulated execution times. The times are based on the assumption that it takes one second to execute any leading or subordinate task and one second to traverse a communication link between neighboring processors in the array.

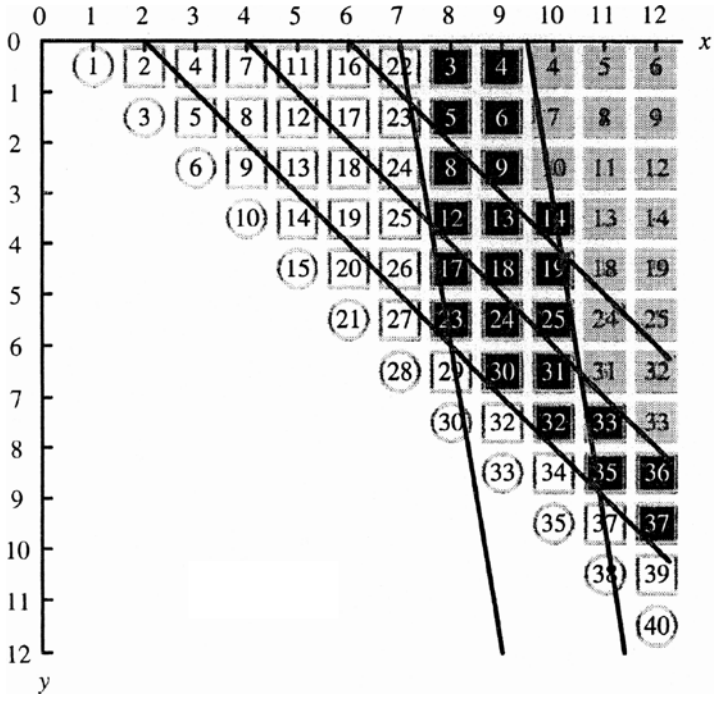


Figure 6.2. Partitioning and sequencing strategy parameterized by w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ for the case $m \geq n$, $n = 24$, $h = 2$, $w = 2$, $\delta_1 = 2.1$, $\delta_2 = 0.0$, $P = 3$ $m_1 = 6.0$, $m_2 = 6.5$, $b_1 = -41.5$, $b_2 = -62.0$, $b'_1 = -2$, $b'_2 = -4$, and $b'_3 = -6$.

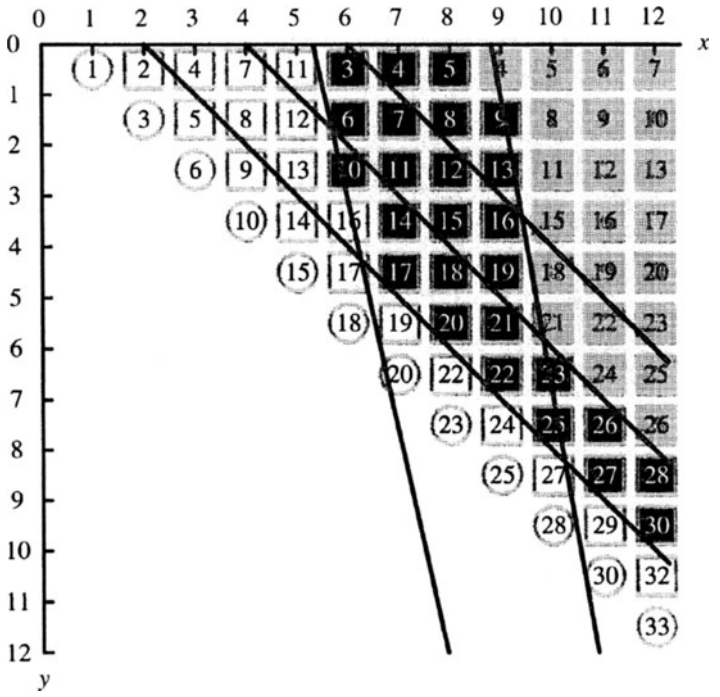


Figure 6.3. Partitioning and sequencing strategy parameterized by w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ for the case $m \geq n = 24$, $h = 2$, $w = 2$, $\delta_1 = 0.0$, $\delta_2 = 0.0$, $P = 3$ $m_1 = 4.3$, $m_2 = 5.8$, $b_1 = -23$, $b_2 = -51$, $b'_1 = -2$, $b'_2 = -4$, and $b'_3 = -6$.

The LB and TP algorithms together partition the task dependency graph into P regions. To satisfy data dependencies among tasks, a communication strategy is devised for multiprocessor execution. Along those lines, the following two procedures manage the asynchronous send and receive operations for the Parameterized Parallel Compact WY algorithm. The first procedure manages those send and receive operations that precede the execution of some tasks, and the second procedure manages those send operations that must take place after the execution of some tasks.

Procedure: ARC (Asynchronous Receive/Send Communication)

Step 1: If TP$(j, i-3/2, p, M, B, B')$ =FALSE then wait to receive columns $[(j-1)h+1 : \min(jh, n)]$ from $p+1$

Step 2: If $p = 1$ then go to Step 6; else continue

Step 3: If TP$(j-1, i-1/2, p, M, B, B')$ =TRUE go to Step 6; else continue

Step 4: Wait to receive Y^k and S^k from $p-1$ where $k=(i-1)h+1$

Step 5: Send Y^k and S^k to $p+1$ where $k=(i-1)h+1$

Step 6: Stop

Procedure: ASC (Asynchronous Send Communication)

Step 1: If TP$(j-1, i-1/2, p, M, B, B')$ = FALSE then send Y^k and S^k to $p+1$ where $k=(i-1)h+1$

Step 2: If TP$(j, i+1/2, p, M, B, B')$ =FALSE and $p>1$ then send columns $[(j-1)h+1 : \min(jh, n)]$ to $p-1$

With the asynchronous communication procedures defined, the Parameterized Parallel Compact WY algorithm is straightforward. The algorithm is presented below where the input parameter $p \in 1, 2, \ldots, P$ is a unique integer identifying the current processor.

Algorithm: PCWY (Parameterized Parallel Compact WY)

Input$(p, A, h, w, \delta_1, \delta, \ldots, \delta_{P-1})$

$[m, n] = \text{dimensions}(A)$

$\tilde{n} = \lceil n/h \rceil$

Compute M, B', and B'' using the LB Procedure

If $p = 1$ then

Distribute A to processors $2, 3, \ldots, P$

For $j = 1$ to $\tilde{n}$

For $i = 1$ to j

If TP$(j, i-1/2, M, B, B')$ = TRUE then

Communicate using the ARC Procedure

If $i = j$ then execute task l_i

Else execute task $s_{i,j}$

Communicate using the ASC Procedure

End If

End For

End For

Else

For $i = 1$ to $\tilde{n}$

For $j = i + 1$ to $\tilde{n}$
If TP$(j, i - 1/2, M, B, B') =$ TRUE then
Communicate using the ARC Procedure
Execute task $s_{i,j}$
Communicate using the ASC Procedure
End If
End For
End For
Output(A)

2. Related Work

Only a handful of authors have proposed parallel orthogonalization algorithms based on the Compact WY algorithm, and they include

Choi et al., 1995

Choi et al., 1995

Baker et al., 1998

The algorithm developed by Choi et al. is at the heart of ScaLAPACK's PDGEQRF algorithm, and the experimental results in Section 3 compare the performance characteristics of the PCWY and PDGEQRF algorithms. Baker et al. modified the PDGEQRF algorithm and showed improved performance for matrix dimensions greater than 5000. In this chapter, m and n are restricted to values less than 3000.

3. Experimental Results

Experimental results are presented for the PCWY algorithm in this section. The results compare the execution times of the PCWY for various values of m and n on a 64-processor HP SPP-2000 and on a 128-processor SGI Origin 2000. The sensitivity of these results to variations in parameter settings is also explored.

For comparison purposes, the results include the execution times of two competing parallel algorithms: a shared memory versions of LAPACK's DGEQRF algorithm for the HP and ScaLAPACK's PDGEQRF for the SGI. Both algorithms require a user-specified parameter LWORK, and the recommended value was used. For PDGEQRF, the values of four additional parameters that minimize execution time are determined experimentally. The parameters are the number of P_r rows and P_c columns in the process grid and blocking factors b_r and b_c for controlling block cyclic distribution.

		$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
HP SPP-2000	PCWY Non-blocking MPI	2.99	1.10	0.416	0.100	0.0180
	DGEQRF Shared Memory	3.90	1.19	0.570	0.212	0.0426
SGI Origin 2000	PCWY Non-blocking MPI	2.46	1.32	0.396	0.119	0.0180
	PDGEQRF BLACS	4.11	2.89	0.728	0.260	0.0420

Table 6.1. Execution times in seconds for $h = h^*$, $w = 2$, $\delta_1 = 0$, $\delta_2 = \delta_3 = \ldots = \delta_{P^*-1} = 0.86$, and $P = P^*$ where h^* and P^* are given in Table 6.2.

Because the data distribution schemes employed by PCWY and PDGEQRF are vastly different, the reported execution times for both algorithms include the cost of distributing the matrix A from processor one to the other $P - 1$ processors. For PCWY, this includes the time for processor one to distribute the requisite columns of the matrix A to processors $2, 3, \ldots, P$. The linear array is assumed to have no broadcasting facilities. No such restriction is placed on PDGEQRF. The execution time for PDGEQRF includes the time to broadcast the matrix A from processor one to the other processors and the time to pack the data on the local processor for two-dimensional block cyclic execution. Even though the upper triangular result is stored entirely on processor one for PCWY and is distributed across P processors in packed format for PDGEQRF, the costs associated with distributing or gathering the final result for PCWY or PDGEQRF is not included. The true nature of this cost would depend on the processing to follow.

The number of combinations of the parameters that satisfy the constraints $1 \leq h \leq n$, $1 \leq hw \leq n$, and $0 \leq \delta_i \leq P$ for $i = 1, 2, \ldots, P - 1$ is infinite and devising an efficient procedure for determining near optimal combinations is beyond the scope of this book. Nonetheless, some definitive patterns have emerged in the results that can guide the end user in selecting combinations that produce high levels of performance. These patterns are discussed in the following paragraphs.

Table 6.1 presents the execution times of the PCWY algorithm, DGEQRF, and PDGEQRF. For experimentally determined optimal h (h^*) and P (P^*), and for constant w and $\delta_1, \delta_2, \ldots, \delta_{P^*-1}$, PCWY outperforms the competing algorithms for all values of m and n on both the HP and the SGI. For

		$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
HP SPP-2000	P^*	53	53	12	11	4
	h^*	8	8	10	22	10
SGI Origin 2000	P^*	62	66	12	12	4
	h^*	8	6	12	16	4

Table 6.2. Experimentally determined optimal values of h and P for $w = 2$, $\delta_1 = 0$, and $\delta_2 = \delta_3 = \ldots = \delta_{P^*-1} = 0.86$.

$500 \geq m \geq n$, PCWY outperforms competing algorithms by a wide margin. The margin narrows for larger problems and is essentially negligible on the HP for $m = n = 1500$. Chapter 5 showed that shared memory outperforms message passing on the HP for $P \leq 16$ and that $m = n = 1500$ is small enough of a problem to make the benefits of using more than 16 processors either in message passing or shared memory marginal. The corresponding parameter settings for PCWY on the HP and SGI are presented in Table 6.2.

	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
HP SPP-2000	2.99	1.10	0.457	0.136	0.0196
SGI Origin 2000	2.46	1.38	0.422	0.144	0.0194

Table 6.3. Execution times for $h = 8$, $w = 2$, $\delta_1 = 0.0$, $\delta_2 = \delta_3 = \ldots = \delta_{P^*-1} = 0.86$, and $P = P^*$ where P^* is taken from Table 6.2.

The parameter h controls the aggregation of Householder reflections and the composition of two types of tasks. By adjusting the parameter, the end user can manipulate the performance tradeoff between finding enough fine-grain parallelism to keep the matrix-matrix multiplication kernels operating at peak efficiency and finding enough coarse-grain parallelism to evenly balance the load among P processors. To a lesser extent, multiprocessor parameters w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$ affect this tradeoff by adjusting the partitioning strategy and therefore the load imbalance among P processors. The discussion of the sensitivity to variations in parameter settings is limited to the parameters h and P. As it turns out constant $w = 2$, $\delta_1 = 0.0$, and $\delta_2 = \ldots = \delta_{P-1} =$

	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
HP SPP-2000	3.31	1.69	0.465	0.188	0.0234
SGI Origin 2000	2.58	NA	0.396	0.127	0.0231

Table 6.4. Execution times for $h = 12$, $w = 2$, $\delta_1 = 0.0$, $\delta_2 = \delta_3 = \ldots = \delta_{P^*-1} = 0.86$, and $P = P^*$ where P^* is taken from Table 6.2.

0.86 achieves near optimal results for all m and n. Despite the fact that no quantitative model to characterize the performance tradeoff controlled by h and P exists, a handful of qualitative observations are gleaned to guide the end user in choosing a parameter combination that produces high levels of performance.

	P	$m = 3000$ $n = 1500$	$m = 1500$ $n = 1500$	$m = 1500$ $n = 500$	$m = 500$ $n = 500$	$m = 500$ $n = 100$
HP SPP-2000	1	43.30	17.30	2.70	0.630	0.0426
	2	35.13	12.10	1.66	0.374	0.0247
	4	22.56	7.45	0.979	0.225	0.0196
	8	13.46	4.29	0.568	0.161	NA
	16	8.15	2.56	0.563	0.148	NA
	32	4.46	1.47	0.479	0.141	NA
SGI Origin 2000	1	41.70	12.7	1.81	0.377	0.0440
	2	18.54	7.17	1.09	0.286	0.0210
	4	10.98	4.43	0.714	0.193	0.0194
	8	8.05	2.66	0.499	0.148	NA
	16	4.65	1.82	0.405	0.141	NA
	32	3.42	1.49	0.438	0.162	NA
	64	2.95	1.44	NA	NA	NA

Table 6.5. Execution times as a function P for $h = 8$, $w = 2$, $\delta_1 = 0.0$, $\delta_2 = \delta_3 = \ldots = \delta_P = 0.86$ where h^* is the experimentally determined optimal value of the parameter h for each experiment.

From Tables 6.1, 6.3, and 6.4, the parameter setting $h = 8$ obtains near optimal performance for all values of m and n, constant values of w and $\delta_1, \delta_2, \ldots, \delta_{P-1}$, and $P = P^*$. The fact that execution times associated with $h = 12$ are as much as 50% slower in some cases than the execution times associated with $h = 8$ leads us to conclude that the benefits of reducing load imbalance outweigh the benefits of improving matrix-matrix multiplication kernel efficiency. Given the importance of load imbalance on selecting optimal values of h, it is not surprising that PCWY performance is sensitive to P as shown in Table 6.5. The parameters h and w provide the following upper bound on P:

$$P \leq \min(P, \lfloor(\lceil n/h \rceil + w - 1/2)/w \rfloor).$$

Unfortunately using as many processors as possible does not necessarily result in the lowest execution time, as is the case for the SGI when $m = n = 500$. In practice, the optimal number of processors should be determined experimentally for $h = 8$, $w = 2$, $\delta_1 = 0$, and $\delta_2 = \delta_3 = \ldots = \delta_{P-1} = 0.86$. If execution time is absolutely critical, as is often the case when using parallel computers, optimal values of other parameters should also be determined experimentally to obtain peak performance.

Chapter 7

CASE STUDY 3: PARALLEL MATRIX BIDIAGONALIZATION

Bidiagonal factorization is the first step in solution procedures for computing the singular values of a matrix $A \in \mathcal{R}^{m \times n}$ (Demmel and Kahan, 1990; Golub and Van Loan, 1989). The factorization is defined as

$$B = U^T A V \tag{7.1}$$

where $B \in \mathcal{R}^{m \times n}$, with $B_{i,j} = 0$ for $i > j$ and $i < j + 1$, and $U \in \mathcal{R}^{m \times m}$ and $V \in \mathcal{R}^{n \times n}$ are orthogonal matrices and only computed if the left and right eigenvectors are needed. Bidiagonalization typically dominates the execution time. For instance, to compute the singular values of a 1000×1000 matrix using LAPACK on a single processor of an HP V2500 requires 17.2 seconds. Of these, 15.5 seconds are spent computing the bidiagonal factorization.

This chapter discusses the design, implementation, and performance of a parameterized, parallel, two-Phase bidiagonalization algorithm that first reduces a dense matrix to an upper triangular matrix with h superdiagonals and then reduces this banded matrix to bidiagonal form. The user-defined blocking parameter h controls the aggregation of Householder transformations. Aggregated transformations are applied in block fashion using the Compact WY representation. This representation permits the use of matrix multiplication in the block application of Householder transformations and obviates the need for explicit superscalar and memory hierarchy parameterizations if vendor-tuned matrix multiplication kernels are available. If tuned kernels are not available, then the parameterization procedures described in Chapter 3 can be applied to the algorithm design process. Phase 2 selectively annihilates elements of the upper triangular matrix using Givens rotations to produce the final bidiagonal form.

The motivation for designing algorithms to exploit tuned matrix multiplication kernels stems from the observation that an efficient subroutine can

exploit properties of the memory hierarchies and processor superscalar design to compute a matrix-matrix multiplication faster than it can sequentially compute the component matrix-vector multiplications (Bischof and Van Loan, 1987; Dongarra et al., 1990; Gallivan et al., 1988). This is evidenced on the SGI POWER Challenge, where LAPACK reports an efficiency of 268 Mflops when multiplying two 1000×1000 matrices, but only 41 Mflops when multiplying a 1000×1000 matrix and a 1000 element vector (Anderson et al., 1995).

1. Parallel Matrix Bidiagonalization Algorithm

Aggregating Householder transformations is a well-known technique for introducing matrix-matrix multiplication (Bischof and Van Loan, 1987; Dongarra et al., 1989). However, in the case of the SHB algorithm, Householder vectors on the left u_k and on the right u_{k+1} cannot be aggregated because the vector v_k, that depends on u_k, must be computed and applied before u_{k+1} can be computed. The following two-Phase parallel matrix bidiagonalization (PMB) algorithm circumvents this dependency constraint. The first Phase uses a technique developed by Bischof et al. (1994) to reduce A to a matrix C with upper bandwidth h (that is, a matrix with h superdiagonals) satisfying

$$C = U_1^T A V_1 \tag{7.2}$$

where $C_{i,j} = 0$ for $i > j$ and $i < j + h$, and U_1 and V_1 are orthogonal matrices. The second Phase applies Givens rotations from the left and the right reduce C to

$$B = U_2^T C V_2 \tag{7.3}$$

where $B_{i,j} = 0$ for $i > j$ and $i < j + 1$, and U_2 and V_2 are orthogonal.

1.1 Superscalar and Memory Hierarchy Parameterization

In the SCWY algorithm, the parameter h controls the aggregation of Householder reflections. As the basis for the PMB algorithm, the SCWY algorithm obviates the need for explicit superscalar and memory hierarchy parameterizations. Beginning with $C_0 = A$, the aggregated reflections are applied using the Compact WY representation to compute

$$C^{*\,k+h} = \left(I + \hat{Y}^k \hat{T}^{k\,T} \hat{Y}^{k\,T}\right) C^k \tag{7.4}$$

and

$$C^{k+h} = C^{*\,k+h} \left(I + \check{Y}^k \check{T}^{k\,T} \check{Y}^{k\,T}\right) \tag{7.5}$$

for $k = 0, h, 2h, \ldots, l$ where l is the largest multiple of h that is less than n. The matrices $\hat{Y}^k \in \mathcal{R}^{\tilde{m} \times h}$ and $\hat{T}^k \in \mathcal{R}^{h \times h}$ are determined from C^k

such that columns k to $k+q-1$ of $C^{*\,k+h}$ are zero below the diagonal where $\tilde{m} = m-k$ and $q = \min(h, n-k)$. The matrices $\check{Y}^k \in \mathcal{R}^{h\times\tilde{n}}$ and $\check{T}^k \in \mathcal{R}^{h\times h}$ are determined from $C^{*\,k+h}$ such that rows k to $k+r-1$ of C^{k+h} are zero to the right of the h^{th} superdiagonal where $\tilde{n} = n-k-h$ and $r = \min(h, m-k)$. The matrix $\hat{Y}^k$ is a collection of Householder column vectors whose leading $k-1$ elements are equal to zero. Similarly, $\check{Y}^k$ is a collection of Householder row vectors. Section 2.2 provides a procedure for determining $\hat{T}^k$ and $\hat{Y}^k$; the computation of $\check{T}^k$ and $\check{Y}^k$ is a straightforward extension of this procedure.

Tuned matrix-matrix multiplication kernels are generally register and cache efficient. However, the efficiency depends on the dimensions of the matrices. For Eqs. 7.4 and 7.5, the efficiency of the matrix multiplication operations depends on $m-k$, $n-k$, and h.

1.2 Multiprocessor Parameterization

For multiprocessor execution, the matrix multiplication operations in Eqs. 7.4 and 7.5 are divided into two sets of P independent operations. Specifically, processor p computes columns n_p through $n_p + \hat{n} - 1$ of $C^{*\,k+q}$ for $p = 1, 2, \ldots, P$ where $n_p = k+h+1+(p-1)\hat{n}$ and $\hat{n} = \lceil (n-k-h)/P \rceil$. Likewise, processor p computes rows m_p through $m_p + \hat{m} - 1$ of C^{k+p} for $p = 1, 2, \ldots, P$ where $m_p = k+h+1+(p-1)\hat{m}$ and $\hat{m} = \lceil (m-k-h)/P \rceil$.

A formal description of Phase 1 is given below, where it is assumed that A is an $m \times n$ element array that initially stores the matrix A, and subsequently stores the current iterate $C^{*\,k+p}$ or C^{k+p}.

Algorithm: Phase 1 of the Parallel Matrix Bidiagonalization (PMB1)

Input(A, h)

$[m, n]$=dimensions(A)

$C = A$

$\bar{m} = \min(m-1, n)$

For $k = 1$ to $\bar{m}$ by h

$\hat{n} = \min(k+h-1, \bar{m})$

$\hat{n}_P = \lceil (n-k-h+2)/P \rceil$

Compute $[v_k, v_{k+1}, \ldots, v_{\hat{n}}]$ using the SH algorithm to factor $C_{k:m,k:\hat{n}}$

Compute $\hat{Y}$ and $\hat{T}$ using the CWY algorithm

DO IN PARALLEL

For $p = 1$ to P

$n_l = k + h + (p-1)\hat{n}_P - 1$ and $n_r = \min(n, n_l + \hat{n}_P - 1)$

$C^{*\,k+h}_{k:m,n_l:n_r} = \left(I + \hat{Y}\hat{T}^T\hat{Y}^T\right) C^k_{k:m,n_l:n_r}$

End For

If $(k + h - n + 1 \geq 0)$ then

$\hat{m} = \min(k + 2h - 1, n - 1)$

$\hat{m}_P = \lceil (m - k - h)/P \rceil$

Compute $[u_k, u_{k+1}, \ldots, u_{\hat{m}}]$ using the SH algorithm to factor $C_{k:\hat{m},k+h:n}$

Compute $\check{Y}$ and $\check{T}$ using the CWY algorithm

DO IN PARALLEL

For $p = 1$ to P

$m_l = k + h + (p-1)\hat{m}_P - 1$ and $m_r = \min(n, m_l + \hat{m}_P - 1)$

$C^{k+h}_{m_l:m_r,k+h:n} = C^{*\,k+h}_{m_l:m_r,k+h:n}\left(I + \check{Y}\check{T}^T\check{Y}^T\right)$

End For

End If

End For

Output(C^k)

To reduce the banded matrix C to bidiagonal form, a sequence of Givens column and row rotations is applied to introduce zeros above the superdiagonal of C. The number of rotations required to zero the $(i,j)^{\text{th}}$ element of C decreases as the bandwidth of the submatrix that lies in rows $j-1$ to m and columns $j-1$ to n of C increases. At the completion of Phase 1, the matrix C has a bandwidth of h. To preserve this bandwidth for arbitrary values of i and j, zeros are introduced from right to left and then from top to bottom. More formally, given m, n, h and an h-band matrix stored in an $m \times n$ element array C, Phase 2 is described below:

Algorithm: Phase 2 of the Parallel Matrix Bidiagonalization (PMB2)

Input(C, h)

$[m, n]$ =dimensions(C)

$B = C$

For $i = 1$ to n

For $j = \min(i+h, n)$ to $i+2$ by -1

Apply a sequence of Givens rotations to annihilate element $B_{i,j}$

End For

End For

Output(B)

An example of the behavior of the PMB algorithm in the case $m = 10$, $n = 9$, and $h = 3$ is given in Figures 7.1, 7.2, and 7.3. Figure 7.1 shows the order in which blocks of zeros are introduced in Phase 1. In Phase 2, zeros are created above the superdiagonal in the order shown in Figure 7.2 by applying an alternating sequence of Givens column and row rotations. Figure 7.3 contains the sequence of rotations necessary to introduce the fifth zero in Phase 2, where the boxes contain the array elements modified by the row and column rotations. The elements designated with a circled plus sign show locations where the rotations "fill in" an element, which is initially zero. Subsequent rotations then remove this fill. By completely reducing row i to bidiagonal form before beginning row $i+1$, the "bandwidth" of the updates is maintained, minimizing the number of columns which are twice modified in the course of introducing each zero.

The number of operations required by the PMB algorithm depends on h. Phase one requires $4n^2(m - n/3) - hn(m + n + 8h/3)$ flops, of which half are multiplications and half are additions. For Phase 2, the number of operations required to zero element $C_{i,j}$ is $6\left(\lceil (n+2-j)/(h+1) \rceil - 1\right)(h+4) + 6(n-i) + 18$, of which two-thirds are multiplications and one-third are additions. It is not possible to express exactly the total Phase 2 operation count as a closed form function of m, n, and h. However, when $n \gg h$, the ceiling function may be ignored, and the total operations count is $\left[3n^2(2h^2 + 3h - 5) - 9h^3(n-2)\right]/(h+1)$.

2. Related Work

A number of studies have solution procedures that circumvent this constraint. In particular, Dongarra et al. (1989) alleviate that dependency constraint by decoupling the relationship between u_k and v_k. The authors use the fact that the same orthogonal transformation V that satisfies Eq. (1) can be used to reduce the symmetric matrix A^TA to an $n \times n$ tridiagonal matrix $B^TB = V^TA^TAV$. Their algorithm computes v_k directly from v_{k-1} and the appropriate part of A. Using v_k, selected components of A are updated, and u_k is determined. After aggregating multiple left and right transformations, the remainder of A is updated in block fashion. The resulting algorithm requires $4n^2(m - n/3) + hn(5m - n/2 - 3h/2)$ flops, where h is the

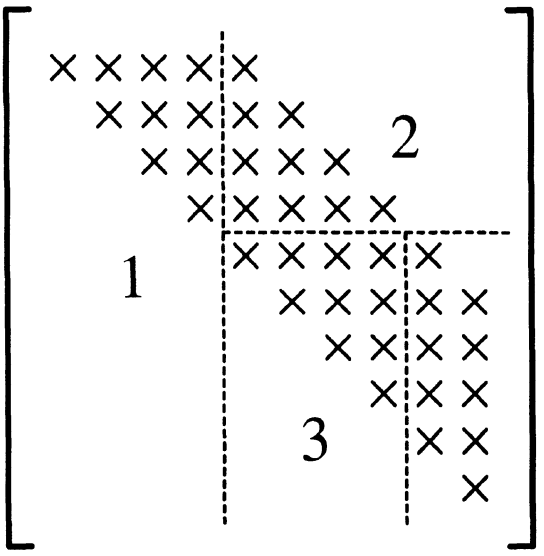


Figure 7.1. Phase 1 of the PMB algorithm: introduction order of blocks of zeros.

block-size parameter that may be tuned for performance. Lawson and Hanson (1974) proposed a two-Phase bidiagonalization (LH) algorithm to reduce the number of operations when $m \gg n$. In Phase 1, A is reduced to an $n \times n$ upper triangular matrix R using QR factorization. In Phase 2, the upper triangular matrix is reduced to bidiagonal form using a method similar to the SHB algorithm. Chan improved on this technique in 1982, proposing that Givens rotations be used for Phase 2 instead of Householder reflections. The resulting R-bidiagonalization algorithm requires $2n^2\,(m + 4n/3)$ flops – a savings over the SBH algorithm, when $m > (5/3)n$. An additional benefit of using QR factorization in Phase 1 is that it permits the use of matrix-matrix multiplication, as in the compact WY algorithm (Schreiber and Van Loan, 1989).

Note that when $h = 1$, the PMB algorithm reduces A directly to a bidiagonal matrix, so Phase 2 is not needed and the resulting algorithm corresponds to the SBH algorithm. Similarly, when $h = n - 1$, the intermediate matrix C is an upper triangular and so the resulting algorithm corresponds to Chan's algorithm. Since the PMB algorithm contains the SBH and Chan's algorithms as special cases, one can optimize with respect to h to obtain an algorithm which is guaranteed to perform at least as well as the best of these two algorithms.

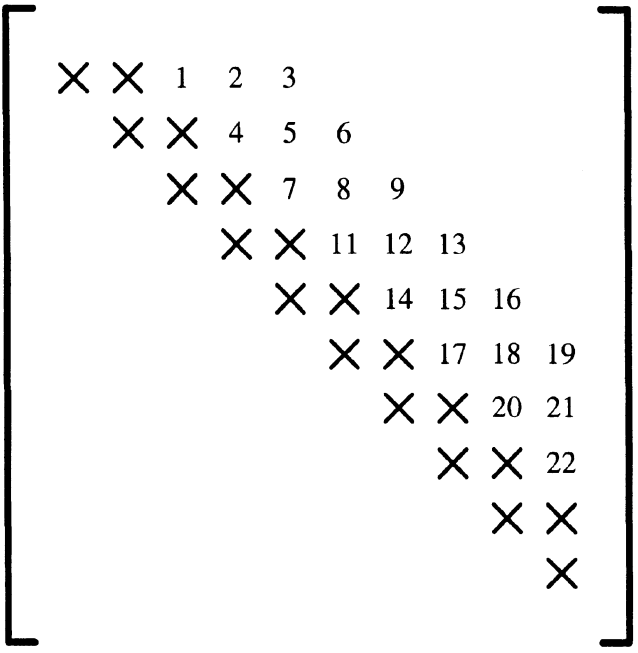


Figure 7.2. Phase 2 of the PMB algorithm: introduction order of zeros.

A closely related problem to matrix bidiagonalization is the reduction of a symmetric matrix to tridiagonal form. Unfortunately, the dependency constraints preventing the aggregation of Householder transformations in the case of bidiagonalization are also present here. Bischof et al. (1994) circumvent this problem by first reducing the symmetric matrix to a symmetric banded form and then reducing the banded matrix to tridiagonal form. This two-Phase process permits the introduction of matrix-matrix multiplication in Phase 1.

3. Experimental Results

In this section, the execution times of a PMB-based SVD (PMB-SVD) algorithm and the standard for computing the singular values of a matrix, LAPACK's DGESVD algorithm, are compared on a SGI Origin 2000. Execution times for PMB-SVD include the execution time of LAPACK's DBDSQR for converting bidiagonal matrices to diagonal form. The PMB algorithm employs the BLAS-3 (Dongarra et al., 1990) subroutine DGEMM for all of the matrix-matrix multiplication operations. In addition, this section explores the

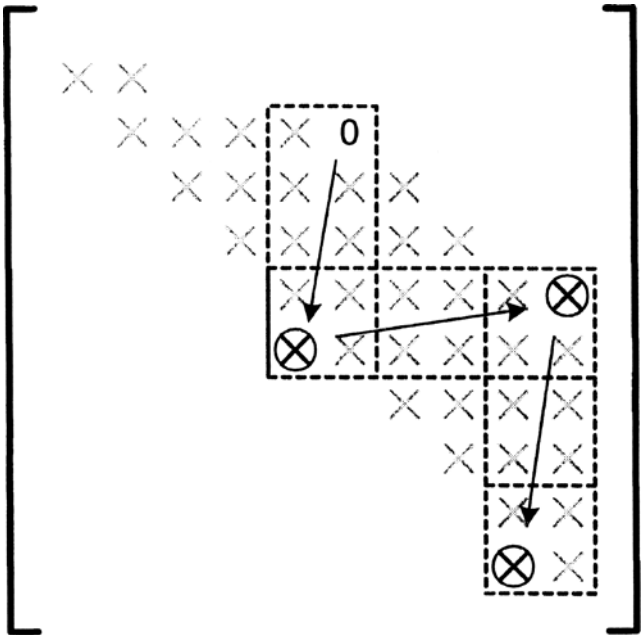


Figure 7.3. Phase 2 of the PMB algorithm: rotation sequence necessary to introduce the fifth zero.

performance characteristics of the PMB algorithm as function of the block parameter h and the number of processors P on a HP V2500 as well.

Figures 7.4 compare the execution times of the PMB-SVD and DGESVD algorithms for various problem dimensions on a single processor as a function of the parameter h. Since h is a tuning parameter for the PMB algorithm, it has no effect on the execution times of the DGESVD algorithm. For $4 < h \leq 170$, PMB outperforms DGESVD on the SGI by as much 30% in all cases. This is attributed to the fact that no more than 50% of the operations in DGESVD are matrix-matrix multiplication (Anderson et al., 1995), whereas the percentage of matrix-matrix multiplication in the PMB algorithm approaches 100% as h is increased. Unfortunately, Phase 2 of the PMB algorithm has no matrix multiplication operations. While the primary role of the parameter h is to control the aggregation of Householder transformations on the left and the right, the parameter also controls the relative amount of work performed in Phases 1 and 2. For large h, the matrix multiplication operations in Phase 1 are more efficient, but the number of elements that

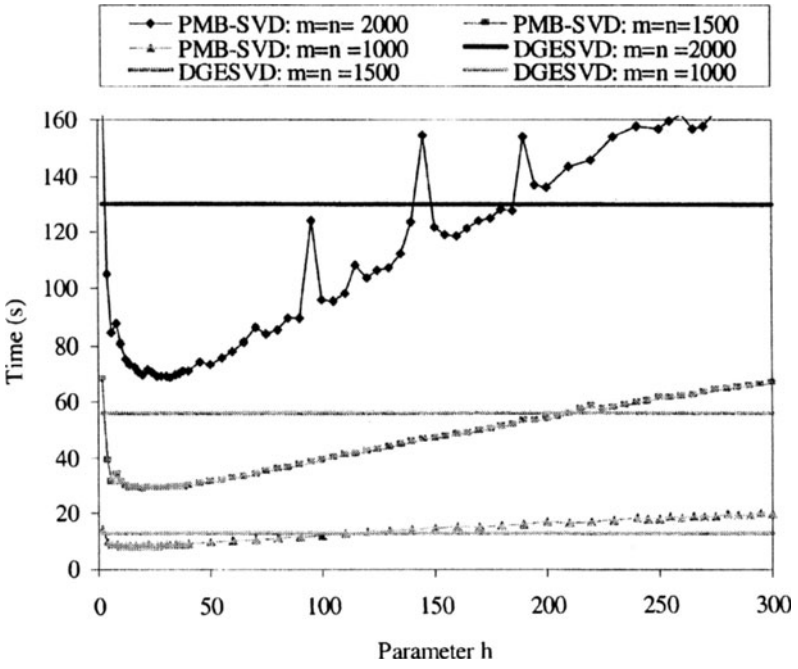


Figure 7.4. Execution times as function of the parameter h on the SGI Origin 2000 for various problems dimensions.

need to be annihilated in Phase 2 by Givens rotations is also large. For small h, the matrix multiplication operations in Phase 1 are less efficient, but the number of elements to be annihilated in Phase 2 is much smaller. Recall that Phase 2 is composed primarily of Givens rotations, and as was discussed in Chapter 4, standard Givens rotations are not superscalar efficient. In addition, the rotations are applied to relatively small vectors of length $h + 1$ or less, further contributing to the inefficiency of Phase 2 in comparison to Phase 1. The resulting performance tradeoff as a function of the parameter h between the two Phases is evident in Figures 7.5 and 7.4.

Figures 7.6 and 7.7 compare the execution times of the PMB for various problem dimensions as a function of the parameter P. For each value of the parameter P and square matrix dimension, h was set to an experimentally determined optimal value. Not surprisingly, the benefits of increasing P diminish rapidly. Parallelism in this algorithm is limited to Phase 1. The execution in Phase 2 is unaffected by the parameter P and begins to dominate the total execution time as P increases. In addition, as P increases, the size

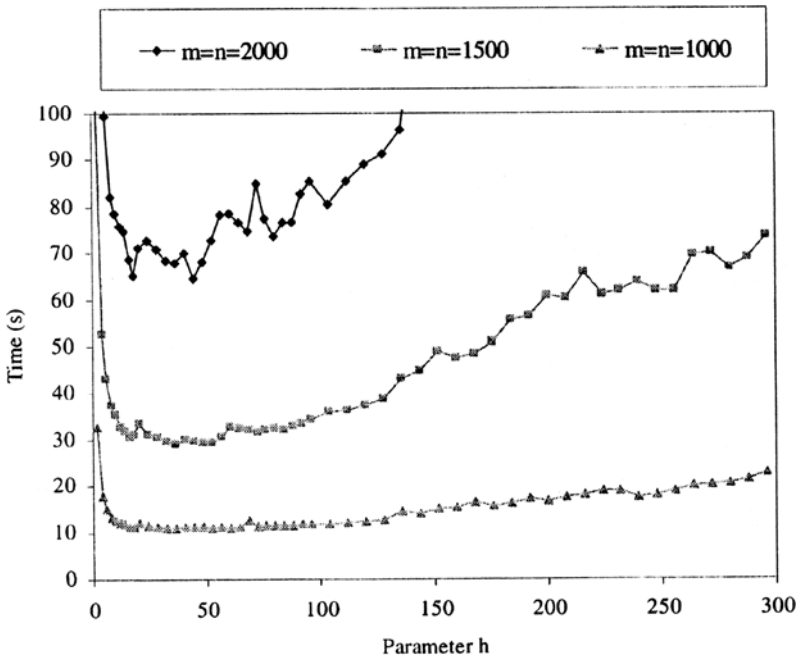


Figure 7.5. Execution times as function of the parameter h on the HP V2500 for various problems dimensions.

of the matrix multiplication operations decrease and so does the efficiency of these operations.

By manipulating the parameters, this chapter has shown that high levels of performance can be extracted from two machines, and that by distributing matrix multiplication operations across P processors some additional levels of performance can be extracted. On a single processor, the PMB algorithm outperformed LAPACK's DGESVD in all cases. Unfortunately, there are no vendor-tuned parallel algorithms available for these machines to compare with the PMB algorithm on multiple processors.

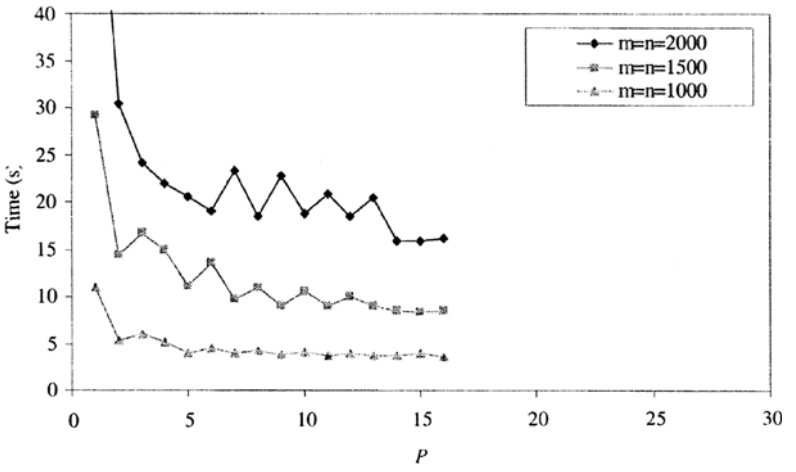


Figure 7.6. Execution times as function of the number of processors P on the HP V2500 for various problem dimensions.

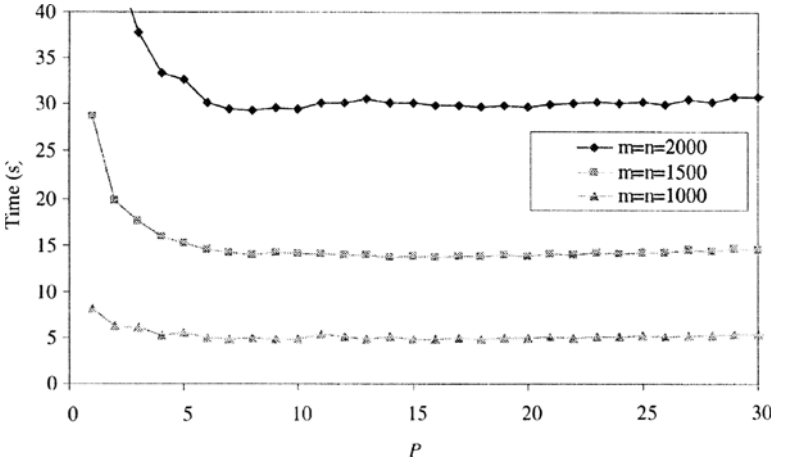


Figure 7.7. Execution times as function of the number of processors P on the SGI Origin 2000 for various problem dimensions.

Chapter 8

CONCLUSION

Despite decades of research in industry and academia, parallel algorithm design largely remains an art form. The systematic steps in the Parallel Algorithm Synthesis Procedure provide algorithm designers with a framework for mastering this art form. The case studies demonstrate that the synthesis procedure is a road map for designing reusable building blocks of adaptable, scalable software components from which high performance signal processing applications can be constructed. The semi-systematic process for introducing parameters to control the partitioning and scheduling of computation and communication allows algorithm designers to simultaneously reap the benefits of efficiency and portability. The parameters are essentially a convenient representation of a large class of algorithms. They allow the algorithm designer to optimize over a large class of algorithms, enhancing both portability and efficiency.

While the synthesis methodology encompasses multiple layers of parameterization, the varying complexities of the case studies demonstrate that not all parameterizations are necessary to achieve high levels of performance. In some cases, tuned kernels can take on the role of a primitive and obviate the need for certain parameterizations. Tuned kernels make efficient use of the memory hierarchy, as is the case with the tuned matrix multiplication kernels. Despite the fact that some parameterizations may not be necessary, it is our experience that the remaining parameterizations should not be applied out of order. The underlying ordering of the parameterizations is crucial to the success of the synthesis procedure. The poor performance of some optimizing compiler technology is rooted in its failure to first tune for superscalar performance, then memory hierarchy performance, and finally multiprocessor performance.

Besides the Parallel Algorithm Synthesis Procedure, another important facet of this work is the presentation of three case studies on parallel matrix factorization. The case studies demonstrate the efficacy of the synthesis procedure with the development of three parameterized parallel algorithms that outperform competing algorithms for every scenario examined. In addition by exploring the performance characteristics of the parameter space, the algorithms shed some light on potential improvements vendors can make to parallel computer architectures.

The most promising directions for future work lie in the investigation of more complex algorithms, partial automation of the parameterization procedures in Chapter 3, and the expansion of the procedure to include the parameterization of the tradeoff between static data distribution schemes (cyclic or otherwise), and schemes that attempt to evenly distribute the load among processors. The latter would prove very useful in helping algorithm designers exploit the performance tradeoffs between these schemes.

References

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., , and Sorensen, D. (1995). *LAPACK Users' Guide, Second Edition*. SIAM Publications, Philadelphia, PA, USA.

Andersson, S., Bell, R., Hague, J., Holthoff, H., Mayes, P., Nakano, J., Shieh, D., and Tuccillo, J. (1998). Rs/6000 scientific and technical computing: Power3 introduction and tuning guide. RedBooks SG24-5155-00, IBM Corporation, International Technical Support Organization, 11400 Burnet Road, Austin, TX 78758-3493.

Badia, J. M., Quintana, G., and Vidal, A. M. (1994). Efficient parallel qr decomposition on a network of transputers. In Becker, M., Litzler, L., and Tehel, M., editors, *Transputers '94. Proceedings of the International Conference*, pages 247–265, Amsterdam, Netherlands. IOS Press.

Baker, G., Gunnels, J., Morrow, G., Riviere, B., and van de Geijn, R. (1998). Plapack: High performance through high-level abstraction. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 414–422, Lso Alamos, CA. IEEE Comput. Soc.

Bell, G. and van Ingen, C. (1999). Dsm perspective: another point of view. *Proceedings of the IEEE*, 87(3):412–417.

Bischof, C., Lang, B., and Sun, X. (1994). Parallel tridiagonalization through two-step band reduction. Technical Report ANL-MCS-P412-0194, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA.

Bischof, C. and Van Loan, C. (1987). The WY representation for products of Householder matrices. *SIAM J. Sci. Comput.*, 8(1):s2–s13.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). *ScaLAPACK user's guide*. SIAM Publications, Philadelphia, PA, USA.

Bodin, F. and O'Boyle, M. (1996). A compiler strategy for shared virtual memories. In Szymanski, B. K. and Sinharoy, B., editors, *Proceedings 3rd Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 57–69, Norwell, MA. Kluwer Academic Publishers.

Bova, S. W., Breshears, C. P., Cuicchi, C., Demirbilek, Z., and Gabb, H. (1999). Nesting openmp in an mpi application. In Olariu, S. and Wu, J., editors, *Proceedings of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*, pages 566–571, Cary, NC, USA. ISCA.

Carrig, J. J. and Meyer, G. G. L. (1997). Efficient householder qr factorization for superscalar processors. *ACM Trans. Math. Software*, 23(3):362–378.

Carrig, J. J. and Meyer, G. G. L. (1999). A parameterized ordering for cache-, register- and pipeline-efficient givens qr decomposition. *Advances in Comp. Math.*, 10:97–113.

Chamberlain, R. M. and Powell, M. J. D. (1988). Qr factorization for linear least squares problems on the hypercube. *IMA Journal of Numerical Analysis*, 8(4):401–413.

Choi, J., Dongarra, J. J., and Walker, D. W. (1995). The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numer. Algorithms*, 10:379–399.

Chu, E. and George, A. (1989). Qr factorization of a dense matrix on a shared-memory multiprocessor. *Parallel Computing*, 11(1):55–71.

Cosnard, M., Daoudi, M., Muller, J. M., and Robert, Y. (1986). On parallel and systolic givens factorizations of dense matrices. In et al., M. C., editor, *Parallel Algorithms and Architectures*, pages 245–258, Amsterdam, Netherlands, North Holland. Elsevier Science Pub. Co.

Dagnum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

Darte, A. and Vivien, F. (1997). Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496.

Demmel, J. and Kahan, W. (1990). Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Sta. Comput.*, 11(5):873–912.

Desprez, F., Dongarra, J., Rastello, F., and Robert, Y. (1998). Determining the idle time of a tiling: new results. *J. Inform. Sci. Engrg.*, 14(1):167–190.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16(1):1–17.

Dongarra, J. J., Sameh, A. H., and Sorensen, D. C. (1986). Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, 3:25–34.

Dongarra, J. J., Sorensen, D. C., and Hammarling, S. J. (1989). Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math.*, 27:215–227.

Dunn, I. N. and Meyer, G. G. L. (2002). Qr factorization for shared memory and message passing. *Parallel Computing*, 28:1507–1530.

Gallivan, K., Jalby, W., Meier, U., and Sameh, A. H. (1988). Impact of hierarchical memory systems on linear algebra algorithm design. *Internat. J. Supercomputer Appl.*, 2(1):12–48.

Golub, G. and Kahan, W. (1965). Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Numer. Anal. Ser. B2*, pages 205–224.

Golub, G. H. and Van Loan, C. F. (1989). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition.

Jainandunsing, K. and Deprettere, E. F. (1989). A new class of parallel algorithms for solving systems of linear equations. *SIAM J. Sci. Stat. Comput.*, 10(5):880–912.

Judge, A., Nixon, P., Tangney, B., Weber, S., and Gahill, V. (1999). Distributed shared memory. In Buyya, R., editor, *High performance cluster computing: Architectures and systems*, volume 1, pages 409–438. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Larriba-Pey, J. L., Valero-Garcia, M., and Navarro, J. J. (1992). Parallel qr decomposition on the suprenum multiprocessor. In Valero, M., Onate, E., Jane, M., Larriba, J. L., and Suarez, B., editors, *Parallel Computing and Transputer Applications*, pages 167–176, Barcleona, Spain. CIMNE.

Lawson, C. L., Hanson, R., and Kincaid, D. (1979). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5(3):308–323.

Lim, A. W. and Lam, M. S. (1998). Maximizing parallelism and minimizing synchronization with affine transforms. *Parallel Computing*, 24(3–4).

Lord, R. E., Kowalik, J. S., and Kumar, S. P. (1983). Solving linear algebraic equations on an mimd computer. *J. ACM*, 30(1):103–117.

Louka, B. and Tchuente, M. (1988). Givens elimination on systolic arrays. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 638–647, New York, NY. ACM.

Lucka, M., Vajtersic, M., and Viktorinova, E. (1996). Massively parallel poisson and qr factorization solvers. *Computers and Mathematics with Applications*, 31(4-5):19–26.

Maslennikow, O., Kaniewski, J., and Wyrzykowski, R. (1998). Fault-tolerant qr-decomposition algorithm and its parallel implementation. In Pritchard, D. and Reeve, J., editors, *Proceedings of the 4th International Euro-Par Conference*, pages 798–803, Berlin, Germany. Springer-Verlag.

Message Passing Interface Forum (1997). Mpi: A message passing interface standard. Technical report, Univ. of Tennessee, Knoxville, TN.

Meyer, G. G. L. and Pascale, M. (1995). A family of parallel qr factorization algorithms. *Concurrency, Practice and Experience*, 8(6):461.

Modi, J. J. and Clarke, R. M. B. (1984). An alternate givens ordering. *Numer. Math.*, 43(1):83–90.

OpenMP Architecture Review Board (1999). Openmp fortran application program interface. Technical report.

Park, N., Prasanna, V. K., and Raghavendra, C. S. (1999). Efficient algorithms for block-cyclic array redistribution between processor sets. *IEEE Trans. on Parallel and Distributed Computing*, 10(12):1217–1240.

Petit, A. P. and Dongarra, J. J. (1999). Algorithm redistribution methods for block-cyclic decompositions. *IEEE Trans. on Parallel and Distributed Systems*, 10(12):1201–1216.

Porta, T. (1988). Using givens rotations to solve dense linear systems on the hypercube. In Kartashev, L. P. and Kartashev, S. I., editors, *Proceedings of the Third International Conference on Supercomputing*, volume 2, pages 443–447, St. Petersburgh, FL, USA. International Supercomputing Institute.

Pothen, A. and Raghavan, R. (1989). Distributed orthogonal factorizations: Givens and householder algorithms. *SIAM J. Sci. Stat. Comput.*, 10(6):1113–1114.

Protic, J., Tomasevic, M., and Milutinovic, V. (1996). Distributed shared memory: concepts and systems. *IEEE Parallel Distrib. Technol., Syst. Appl.*, 4(2):63–71.

Sahni, S. and Thanvantri, V. (1996). Performance metrics: Keeping the focus on runtime. *IEEE Parallel Distrib. Technol., Syst. Appl.*, 4(1):43–56.

Sameh, A. and Kuck, D. (1978). On stable parallel linear system solvers. *J. ACM*, 25(1):81–91.

Sarkar, V. (1997). Automatic selection of high-order transformations in the ibm xl fortran compilers. *IBM J. Res. Develop.*, 41(3):233–264.

Schreiber, R. and Van Loan, C. (1989). A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57.

Smith, A. and Suri, S. (2000). Rectangular tiling in multidimensional arrays. *Journal of Algorithms*, 37(2):451–467.

Sun, X.-H. and Gustafson, J. L. (1991). Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109.

Throop, J. (1999). Openmp: shared-memory parallelism from the ashes. *Computer*, 32(5):108–109.

Wilburn, V. C., Hak-Lim, K., and Alexander, W. E. (1996). An algorithm and architecture for the parallel solution of systems of linear equations. In *Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications*, pages 392–398, New York, NY. IEEE.

Wilson, G. V. (1993). The history of the development parallel computing.

Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Shih-Wei, L., Chau-Wen, T., Hall, M. W., Lam, M. S., and Hennessy, J. L. (1994). Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37.

Wolf, M. E. and Lam, M. S. (1991). A loop transformation theory and algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471.

Wolfe, M. (1996a). Parallelizing compilers. *ACM Computing Surveys*, 28(1):261–262.

Wolfe, M. J. (1996b). *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, Redwood City, CA, USA.

Wright, K. (1991). Parallel algorithms for qr decomposition on a shared memory multiprocessor. *Parallel Computing*, 17(6-7):779–790.

Index