Chapter 1

Introduction

Matrix computations are ubiquitous. They arise in the implementations of almost all scientific and engineering applications. Due to the physical properties of these problems, the resulting matrices often have complicated structure. That is, the values in the matrices have some regular pattern in their organization. Although it is well known that exploitation of this structure can yield a major increase in the efficiency of matrix computations, current languages offer little direct support for defining computations over the kinds of structured matrices that arise in modern applications. This dissertation investigates both the formal and practical issues involved in providing powerful language and compiler capabilities for simplifying the generation of efficient sequential and parallel implementations for computations over complexly structured matrices. The result is a technology that enables rapid specification without sacrificing high performance.

This chapter presents the dissertation framework. It begins with the introductory material necessary for understanding the nature of matrix computation. Next, a brief survey describes current methods for implementing structured matrix computations. This provides the context for the problem statement and our approach. The following section presents the key research issues and how they are solved in the approach. The next section discusses the contributions of the dissertation, and the final section describes the structure of the remainder of the dissertation.

1.1 Perspective

Scientific and engineering applications generate some of the most important uses of computing resources. These applications range from the macroscopic modeling of the aerody-

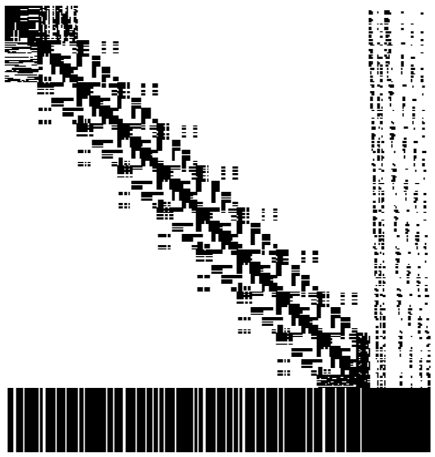


Figure 1-1. A complexly structured matrix. The black regions represent the non-zero values of the matrix.

namics of airplanes and the acoustical properties of submarines to the microscopic world of quantum mechanics and polymer dynamics, and from the practical real-time areas of graphics rendering and power systems to the exotic astrophysics of colliding black holes. But regardless of whether the solution method is partial differential equations discretized by a finite difference/element method or some other mathematical method the implementation is almost invariably an array computation and, more specifically, a matrix computation.

The matrices that arise in these applications are distinguished by the fact that they are often *structured*. In matrix algebra, a number of useful structures have been codified in the form of matrix properties. These properties include such well known concepts as upper triangular, diagonal, symmetric, zero, pentadiagonal, positive definite, and identity. Unfortunately, many matrices that arise in modern applications do not fit so neatly into this simple type system.

Consider the structure of the matrix shown in Figure 1-1. The “structure” of this matrix is far too complicated to fit into any simple concept of matrix type. Yet there does exist some describable regularity in the organization of the values. Moreover, it is well known, that for the sake of efficiency this structure must somehow be described. Exploiting matrix structure is crucial since it can potentially be used to reduce both the number of data elements that must be represented and the amount of computation that needs to be performed. In the actual matrix in Figure 1-1, of the over one million entries in the matrix, less than 15% of them are non-zero. This suggests that by explicitly describing the structure, it should be possible to obtain substantial representational advantages. Similarly, if we examine the kinds of matrix operations that are typically performed, matrix multiplication, linear solve, decomposition (LU

and Cholesky), and eigenvalue determination (QR and QZ) for example, we find that they are of order n^3. As a result, avoiding unnecessary operations can yield a tremendous computational advantage.

In addition to being structured, these matrices are often massive; it is not unusual for there to be in excess of hundreds of thousands of unknowns. Therefore, it may not be possible to achieve reasonable throughput on a sequential architecture. The inevitable conclusion is that parallel implementations are necessary.

1.2 Current Approaches

Currently available matrix languages (and packages) do not provide adequate support for the matrices that occur in modern applications. Languages such as APL [17] were not intended to capture structured matrices. In comparison, numerical libraries, such as BLAS [5] and LAPACK [1], support some simple concepts of typed matrices; their homogeneous interfaces provide access to highly-tuned routines across a wide range of architectures. Unfortunately, the concept of matrix type in these packages is not powerful enough to directly represent the kinds of matrix structures that commonly arise. Therefore, to obtain the desired performance, a user may be forced to code large portions of the application by hand.

1.3 Problem Statement and Approach

This dissertation investigates language and compiler support for complicated, large-scale matrix computations that arise in modern scientific applications. Specifically, we develop theoretical and practical models of both language concepts for capturing the semantics of complexly structured matrices, and compiler mechanisms for exploiting these semantics in the setting of sequential and parallel execution environments.

The central concept is recursive matrices - matrices whose elements are themselves matrices. More accurately, they are objects that represent some hierarchical structuring of a matrix. These *hierarchical matrix structures* serve as a framework for capturing the semantics of matrices. They provide a methodology for viewing complicated matrices as compositions of simpler matrices. Matrix operations over hierarchical matrix structures are viewed in the same manner --- recursively. An implementation of a matrix operation is constructed by composing sub-implementations on a level by level basis over the hierarchical matrix structures.

To gain a better intuitive understanding of the hierarchical view, we will consider some of steps involved in constructing hierarchical structuring of the matrix previously shown in Figure 1-1, and the nature of a recursive implementation of a matrix operation over this structure.

1.4 Example

Figure 1-1 shows the non-zero values associated with a submarine acoustics problem [23]. The purpose of the problem is to model the propagation of waves through the fluid surrounding the submarine. Unfortunately, the computational scientist faces the difficulty that the partially-sparse and partially-dense linear system represented by the matrix is not efficiently solvable with typical methods since the matrix structure cannot be captured by one of the well-known matrix types. If we attempt to elucidate this structure, we discover that it arises due to a complex interaction among the modeling method, the physical object being modeled, and the mapping to the matrix representation. But we find by viewing the matrix hierarchically that these interactions can be teased apart and a describable structure emerges.

At the highest level, the matrix can be viewed as being composed of four sub-regions, A, B, C, and D, as shown in Figure 1-1 (a). This partitioning naturally arises because there are in fact four different techniques involved in the model: region A models the propagation of waves through the shell of the submarine, region D models the propagation of waves through the fluid layer next to the submarine, region B models the propagation of waves from the submarine to the fluid, and region C models the propagation of waves from the fluid to the submarine. If we examine the next level of description, we find that each of these sub-regions has further structuring. For example, region A can be viewed as a tridiagonal structure as shown in Figure 1-2 (b). This structure arises due to the modeling of the physical structure of the submarine as being sliced into rings; only adjacent rings interact, leading to a tridiagonal structure. Again, each of the elements of matrix region A can have further hierarchical structure. We will now examine the effect of structure on implementing an operation over this matrix.

The LU decomposition[1] (just LU for short) is a matrix operation that is often used in solving systems of equations. It is well known that an LU can be performed recursively; the common algorithm is to perform an LU on successive elements on the diagonal, moving from top-left to bottom-right, updating the remainder of the matrix after each step. For our example, the first step in applying an LU to the whole matrix is to perform an LU on region A. Since we know that this region is tridiagonal, we can choose an algorithm that specifically exploits this type information; this will lead to a more efficient execution. This LU operation will invoke other LU's over the elements of region A. Due to the types of these elements, other type-specific LU algorithms may be applicable.

This example displays some advantages of hierarchical thinking. Specifically, it exposes some of the benefits of using hierarchical composition for describing both the structure of matrices and the manner of performing matrix operations over them. For describing matrix structure, this approach provides the ability to capture the actual matrix properties on an element-by-element basis (an example is the tridiagonal structure of region A). For implement-

[1] An LU decomposition is a matrix operation that factors a matrix, A, into a (unit) lower triangular matrix, L, and an upper triangular matrix, U, such that L*U=A.

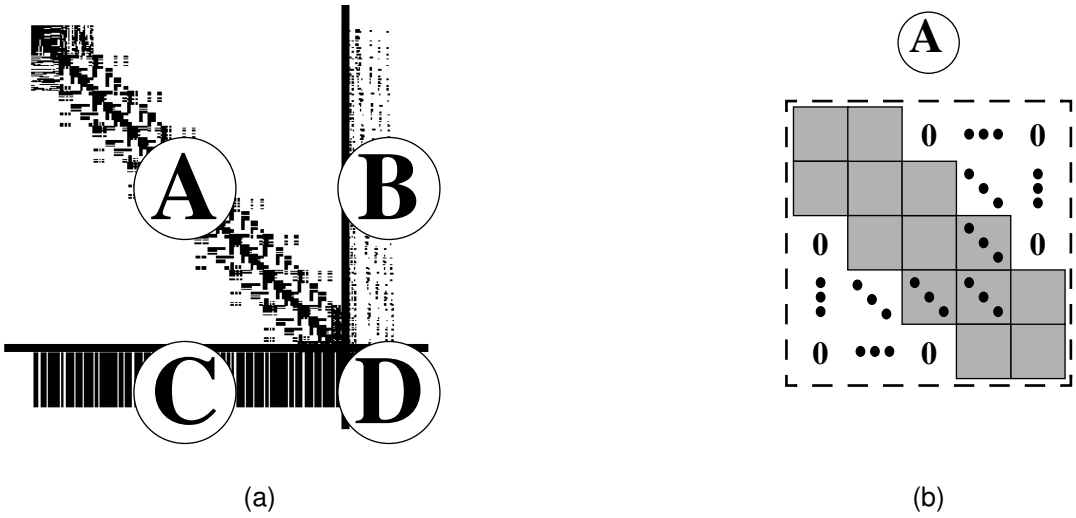


Figure 1-2. A hierarchical matrix structuring. The original matrix partitioned into four regions, (a) and one level of structuring of region A, (b).

ing matrix operations, the hierarchical approach provides opportunities for exploiting type information on an element-by-element basis in order to obtain efficient execution. In addition, hierarchical structures provide an ideal framework for supporting parallel implementations. First, it is possible to annotate the elements of such a structure with information describing their distribution over the separate memories of a distributed memory architecture. Second, functional parallelism can be exposed on an element-by-element, level-by-level basis.

The advantages of the hierarchical approach go beyond just the description of structure. In Chapter 5 we will examine how to view the process that generates matrix structure. This leads to the ability to influence the actual matrix structure. As a result we can choose and describe appropriate structures that are more amenable to efficient sequential and parallel execution of matrix operations.

1.5 Contributions

The contributions of this research include both formal and practical issues involved in the efficient parallel implementation of large-scale, complicated matrix computations. The theory of hierarchical matrices provides a basis for languages and compilers for hierarchical matrix computations. Our MaTRiX+$^+$ prototype supports user access to this capability.

The key contributions of the formal theory include:

- a model for hierarchical matrix structures, and
- a recursive, type-specific implementation strategy for hierarchical matrix operations.

The MaTRiX+$^+$ system contributes to both language and compiler research in matrix computations. Some of the primary features of the language include:

- simple declarative constructs for describing hierarchical structure of application matrices and their data distributions, and
- a simple interface for calling matrix operations.

Some important features of the compilation approach include:

- an extensible, object-oriented, run-time, compilation package for hierarchical matrix computations,
- efficient construction of type-specific implementations for matrix operations, and
- a simple, threaded, data-driven single-program multiple-data (SPMD) parallel model of execution that exploits implicit parallelism.

These contributions make this research an important step towards providing powerful language and compiler support for complicated, large-scale matrix computations. In addition to these specific contributions, the general contribution of the research is the metaphor it provides for solving complicated application problems. This allows scientists to think of the solutions to applications in terms of the problem description rather than in its Fortran implementation.

1.6 Structure of the Dissertation

The following summarizes the contents of each of the remaining chapters of the dissertation.

Chapter 2 presents background information about matrix computations. Also we survey some important research in the area of hierarchical matrix computations.

Chapter 3 defines the formal basis for languages and compilers for hierarchical matrix computations. The focus is the theory for hierarchical matrix structures. In addition, we discuss a strategy for implementing matrix operations over them.

Chapter 4 describes the MaTRiX+$^+$ language prototype. The prototype is a collection of classes written in a parallel dialect of C++ called Compositional C++ (CC++) [3]. Therefore, the constructs provide an object-oriented interface to constructing and using hierarchical matrix structures

Chapter 5 presents the compilation algorithm implementation for the MaTRiX+$^+$ prototype system. The discussion focuses on the issues of extensibility and efficiency. One important aspect of efficiency is the run-time environment. We present a threaded, data-driven SPMD model of execution that exploits parallelism explicitly.

Chapter 6 describes some approaches to coding scientific applications in the MaTRiX+$^+$ system. The example applications include a power systems problem, a finite difference method, and a submarine acoustics problem. The discussion for each example provides executable MaTRiX+$^+$ code. The focus of the discussion is the simplicity of the specifications and the reusability of code and solution conceptualizations.

Chapter 7 presents experimental results. The experiments quantify run-time characteristics of the codes associated with the finite difference method and the power systems application. The aspects that are specifically studied is the are: the computational overhead of the object-oriented implementation approach, the benefits of type information, and the efficacy of the threaded, data-driven SPMD model of execution.

Chapter 8 concludes the dissertation. It provides a brief summary of the research and its major contributions. In addition, some interesting future directions are discussed.

For the reader: This dissertation has been written for a broad audience ranging from computer scientists to computational scientists to application engineers. Although it can read consecutively from Chapter 2 to Chapter 8, there are other reading orders that could be preferable to the reader.

- For the reader who is primarily interested in the relationship between hierarchical types and solving scientific applications, the suggested order is Chapter 2 → Chapter 6 → Chapter 4 → Chapter 5 → Chapter 7 → Chapter 3 → Chapter 8.
- For the reader who is interested in the issues concerning the efficient implementation of scientific applications in object-oriented languages a prefered order is Chapter 2 → Chapter 4→ Chapter 5→ Chapter 6→ Chapter 7 → Chapter 3 → Chapter 8.

Chapter 2

Background and Motivation

Matrix computations provide the cornerstone for modeling scientific applications. Due to the common use of sophisticated computational algorithms, increasingly more complex matrix structures are arising. Although matrices play such an important role, there has very little research in the area of capturing and exploiting semantics of matrices to aid in representation of matrix computations. Instead the focus of much current research has been the role of array operations in parallel computations - theories [20], data structures [2], and models of parallelism [4][10][15][16][18][24] have all been proposed. Unfortunately, treating matrices as two dimensional arrays in these systems neither simplifies the construction of complicated matrix computations nor provides a parallel model of computation that is particularly suited to matrix operations; the semantics of matrix computations require explicit specification.

This chapter discusses some research that addresses the problem of matrix semantics. We begin with a brief introduction to matrix computations. Then we survey some of the more important related work. The majority of the work has been the library efforts of the BLAS [5] and LAPACK [1] projects. Their goal has been to provide architecture-independent efficient numerical routines for special types of matrices. Some additional investigation has been done by Skagestein [26] for "out of core" matrix operations and by Wise [28][29] for quadtree approaches to matrix computations. We will briefly discuss each of these and compare their approaches to that taken in this dissertation.

2.1 Matrix Computation Background

The following discussion provides the basic background for matrix computations. We use a standard notation and style of presentation that is borrowed from [13]; a much more complete discussion can be found there. In general, capital letters such as A, B, L, and U represent matrices and their subscripted lower case counterparts represent specific entries in the matrix. Capital letters in **bold** represent specific matrix structures (types).

2.1.1 Matrix Notations

For $\Re$, the set of real numbers, space of all m-by-n matrices is denoted by $\Re^{mxn}$:

$$A \in \Re^{m \times n} \Leftrightarrow A = (a_{ij}) = \begin{bmatrix} a_{11} & \bullet\bullet\bullet & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \bullet\bullet\bullet & a_{mn} \end{bmatrix} a_{ij} \in \Re.$$

An m-by-n block matrix is a partitioning of a matrix into sub-matrices:

$$A = \begin{bmatrix} A_{11} & \bullet\bullet\bullet & A_{1q} \\ \vdots & & \vdots \\ A_{p1} & \bullet\bullet\bullet & A_{pq} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_p \end{matrix}$$
$$\begin{matrix} n_1 & & n_q \end{matrix}$$

The constraint for legal blocking is that $m_1 + ... + m_p = m$ and $n_1 + ... + n_q = n$ and sub-block A_{ij} has dimensions m_i-by-n_j.

2.1.2 Basic Matrix Operations

The basic matrix operations include the two algebraic ones, addition and multiplication, and two others, transposition and scaling. Matrix addition is defined over $\Re^{mxn}$ x $\Re^{mxn}$ $\rightarrow \Re^{mxn}$ as

$$C = A + B \quad \Rightarrow \quad c_{ij} = a_{ij} + b_{ij}$$

and matrix multiplication is defined over $\Re^{mxr}$ x $\Re^{rxn} \rightarrow \Re^{mxn}$ as

$$C = AB \quad \Rightarrow c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj}.$$

For both of these definitions there are *compatibility* constraints to guarantee appropriate dimensionality of the matrices.

2.1.3 Matrix Structure

We use the term *structure* to indicate an organization over the values of a matrix. The following are some of the well-known structures over a matrix A:

- zero, **Z**, for all i,j $a_{ij} = 0$,
- identity, **I**, for all i,j if i=j then $a_{ij} = 1$ else $a_{ij} = 0$,
- upper triangular, **U**, for all i,j if i>j then $a_{ij} = 0$,
- unit upper triangular, $\mathbf{U_u}$, for all i,j if i>j then $a_{ij} = 0$ and if i=j then $a_{ij} = 1$,
- lower triangular, **L**, for all i,j if i<j then $a_{ij} = 0$,
- unit lower triangular, $\mathbf{L_u}$, for all i,j if i<j then $a_{ij} = 0$ and if i=j then $a_{ij} = 1$,

2.1.4 Structure Algebra

The importance of structure is that it can lead to more efficient matrix computations both representationally and computationally. In addition, structure information can be propagated to the result matrices of operations over structured matrices. This allows structure benefits to be exploited in future operations. The propagation of structure is determined by an algebra of structures. With **X** representing the "don't care" for structure, the following relations are true for matrix addition:

- $\mathbf{Z + X = X}$,
- $\mathbf{U + U = U_u + U_u = U_u + U = U + U_u = U}$,
- $\mathbf{L + L = L_u + L_u = L_u + L = L + L_u = L}$.

Similarly for matrix multiplication, the following relations hold:

- $\mathbf{Z * X = X * Z = Z}$,
- $\mathbf{I * X = X * I = X}$,
- $\mathbf{U * U = U_u * U = U * U_u = U}$,
- $\mathbf{U_u * U_u = U_u}$,
- $\mathbf{L * L = L_u * L = L * L_u = L}$,
- $\mathbf{L_u * L_u = L_u}$.

2.1.5 Other Matrix Operations

The basic matrix operations provide the building blocks for constructing other useful matrix operations such as linear solves, factorizations (also called decompositions), least squares, and eigenvalue determinations. We will look at instances of the first two operations and their relationship to each other.

Given A and B in the equation A * X = B, a linear solve determines the values of X if a solution exists. A factorization is another kind of matrix equation. An LU factorization is the case of solving the equation A = L * U for L, a (unit) lower triangular matrix, and U, an upper

triangular matrix. Since it is straightforward to solve the equations L * X = B and U * X = B, an LU factorization can be used in solving the equation A * X = B in the following manner: solve the equations L * Y = B and U * X = Y such that A * X = L * U * X = B.

2.1.6 Block Algorithms

Block algorithms are algorithms for matrix operations that operate on blocks (matrices) rather than on scalars. Matrix addition and matrix multiplication can be directly defined over (compatible) block matrices by simply substituting blocks for scalars in the definitions given in Section 2.1.2. It is also possible to construct block algorithms for the other matrix operations. Consider the following derivation for a block LU. For a compatibly blocked 2-by-2 matrix A (this can be easily generalized to n-by-n matrices), we obtain the following block matrix equation.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} * \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{21} \end{bmatrix}$$

By performing the multiplication of the L matrix and the U matrix we obtain the equations $L_{11} * U_{11} = A_{11}$, $L_{11} * U_{12} = A_{12}$, $L_{21} * U_{11} = A_{21}$, and $L_{21} * U_{12} + L_{22} * U_{22} = A_{22}$. We observe that because we know how to perform an LU on matrices, the following algorithm will compute a block LU.

[1] factor A_{11} to produce L_{11} and U_{11}

[2] perform triangular solve on $L_{11} * U_{12} = A_{12}$ to produce U_{12}

[3] perform triangular solve on $L_{21} * U_{11} = A_{12}$ to produce L_{21}

[4] compute $A_{22} - L_{21} * U_{12}$ to produce $A_{22}\tilde{}$

[5] factor $A_{22}\tilde{}$ to produce L_{22} and U_{22}

Block algorithms are important in that they provide a model for efficient algorithm design. Coding efficient numerical algorithms requires careful consideration of the memory hierarchy of the architecture. Moving from the external memory to the main memory to the cache and to the registers the storage tends to become smaller and faster. Therefore, it is extremely desirable to keep the majority of the operations using the faster levels. If we consider matrix multiplication (of n-by-n matrices) we find that it performs order n^3 operations over order n^2 data. Matrix addition performs order n^2 operations over order n^2 data. Due to this data movement principle, matrix multiplication can be coded to utilize hierarchical memory effectively, and therefore is an ideal building block for matrix algorithms. In the case of our block LU algorithm, if we block the matrix appropriately so that the majority of the computation is performed in step 4, then we can expect to obtain excellent performance on most modern computers[2]. Block algorithms also expose opportunities for parallel execution.

2.2 Survey of Approaches

Blocks play a central role in most approaches to high-performance computation. For numerical approaches, the computational properties of blocks are of most interest. In the data structure approach, blocks serve as a data structuring concept. We will briefly survey both of these areas of research and discuss their relationship to our approach.

2.2.1 Numerical Approaches

BLAS

The level 3 BLAS [5] were the first step towards widely available numerical block algorithms. They provide a uniform interface to various matrix multiplication and triangular solve operations; these operations are some of the main components of most block algorithms. The idea is that they should serve as computational kernels for the more complicated matrix operations such as solving a system of equations. A number of packages, most notably LAPACK, use the BLAS as building blocks for constructing efficient block algorithms. Due to the standardized interface of the BLAS, it is possible to highly optimize the routines for each target architecture. This has lead to efficient BLAS implementations on a wide variety of architectures.

LAPACK

LAPACK [1] is a collection of routines for solving systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. It is intended to supercede EISPACK [12][27] and LINPACK [8]. The main idea is its recoding of the standard algorithms to use the BLAS numerical kernels; the result is both portability and high-performance. LAPACK also allows the specification of some matrix structures (types) including bidiagonal, tridiagonal, banded, Hessenberg, triangular (unit/non-unit, lower/upper), symmetric, and positive definite. Type information is specified partly in the name of the routine and partly in the parameters.

LAPACK++

LAPACK++ [6] is not actually a numerical package itself, but is rather a C++ wrapper around the LAPACK package. The goal is to provide better accessibility to efficient numerical routines from an object-oriented language. Since a majority of the execution is spent in the actual numerical LAPACK routines, the object-oriented interface incurs very little overhead.

---

[2] In fact, step 1,2,3, and 5 can all be blocked in order to expose more matrix multiplications.

ScaLAPACK

ScaLapack [7] provides a parallel implementation of some of the LAPACK routines. The current approach is to recode LAPACK routines to use a set of BLAS which exploit parallelism in performing operations such as matrix multiplication that are implicit in the BLAS implementation.

2.2.2 Other Numerical Approaches

Dealing with hierarchical memory issues is not a new problem. In the past, core memory (main memory) was extremely limited. Skagestein [26] exploited recursive structuring of matrix computations as a method for efficiently performing out of core matrix operations[3]. The goal was to partition the matrix so that only the parts of the matrix needed during a given phase of a computation were in memory. His approach was to design a hierarchically blocked matrix called a hypermatrix (a tree where all leaves are at the same depth). For a set of matrix structures (triangular, zero, and symmetric positive definite) he constructed recursive block algorithms for matrix operations that operate over hypermatrices.

2.2.3 Discussion

From the standpoint of matrix semantics, the numerical approaches allow a user to select an algorithm based on the types of the input matrices. A user can expect reasonable performance if his matrices fit into the types that are supported by these packages. Unfortunately, the matrices that arise in modern scientific applications often do not fit into this limited type scheme. The following data structuring scheme is one proposed solution to this problem.

2.3 Data Structuring Approach

Quadtrees are hierarchical data structures that provide efficient representations and operations for a number of domains [25]. One special kind of quadtree, a region quadtree is especially useful for matrix representations. A region quadtree divides a matrix into four equal-sized quadrants. If a quadrant is not all zero-valued, then it is in turn divided. This recursive sub-dividing ends when a quadrant contains all zeroes or only a single value.

Wise [28][29] has extensively examined a functional language approach to quadtree matrix computations. He constructs quadtree representations of matrices that gain efficiency through the use of a special *null* pointer for zero quadrants. Since the matrices must have dimensionality that is a power of 2, it is necessary to pad matrices whose dimensions are not a power of 2 with additional zeroes in order to obtain a legal size. In addition he defines matrix operations that utilize the properties of zero and identity matrices during matrix operations to avoid unnecessary computation.

---

[3] This work can only be found in his 1972 dissertation (written in German).

Discussion

The quadtree approach is an automatic means of capturing and exploiting irregularly sparse matrices. Its representation is constructed by explicitly examining the values of the matrix. As such, it does not provide a methodology for capturing matrix semantics in a manner that can be exploited by a compiler. In addition, the concept of describing zeroes is insufficient for a number of matrix types, such as symmetric.

2.4 Summary

Although matrix computations are an important class of computer applications, there does not exist effective methods for representing the complexly structured matrices that arise in modern applications. At one extreme there are the numerical packages that allow users to select efficient implementations for a small class of typed matrices. At the other extreme is the quadtree approach that can manage to generate efficient implementations if the data just happens to organized nicely.

Our approach is semantics based. We provide the user with powerful language constructs for describing the semantics of matrices in terms of hierarchical matrix types. Hierarchical matrix structures, the basis for hierarchical matrix types, are structurally more general than both hypermatrices and quadtrees. Implementations of matrix operations over hierarchical matrix structures are generated through the recursive composition of block algorithms. Efficiency is obtained since the recursion bottoms out in calls to high-performance numerical routines such as those in LAPACK. In addition, hierarchical matrix structures provide a simple framework for integrating information used in partitioning a matrix for a distributed memory environment. The next chapter develops the underlying theory for hierarchical matrix computations.

Chapter 3

A Theory of Hierarchical Matrices

The theory of hierarchical matrices provides a formal basis for the development of matrix languages and compilers for computations over complexly structured matrices. Specifically, it describes models for capturing and exploiting the semantics of the structured matrices that arise in scientific and engineering applications. The key idea is to define the essential semantics of matrices in terms of types. In order to develop a sufficiently powerful notion of type we explore the concept of recursive matrices as a model for hierarchical matrix structures. The appeal of structured types for matrices is that they can simply capture the semantics of structured matrices in a form usable by a compiler.

The hierarchical matrix type system has two goals: it simplifies the specification of complicated matrix computations and it captures semantic information that can be exploited by a compiler during optimization. In both cases the purpose is to gain efficiency in the implementation of the matrix computations. The type information has no effect on the values of the matrices. Therefore, the values in the result matrices are the same whether or not the type information is exploited; Figure 3-1 on page 26 shows a commuting diagram that emphasizes this point.

In this chapter we develop an algebra of hierarchical matrices. It extends the usual matrix algebra by allowing matrices that can have matrices as elements rather than just scalars. Such recursive matrices provide the meta-type for legal hierarchically structured matrices. The first section defines the set of hierarchical matrix structures and some functions over them. The next section develops semantic extensions to this model through the addition of properties, such as triangular and symmetric, to the definition of hierarchical matrix struc-

tures. Section 3 presents the definition of matrix operations and their implementations in the form of matrix algorithms. Section 4 raises some practical issues having to do with the nature of compilers for hierarchically structured matrices. The chapter ends with some concluding remarks. The proofs of the lemmas and theorems can be found in Appendix B.

Notation: Table 3-1 is a brief list of the symbols used in this chapter.

Table 3-1. Symbol definitions

$\therefore$	define
$\mid$	set comprehension
$\Rightarrow$	total function
$\xrightarrow[f]{}$	function with finite domain
$\rightarrow$	partial function
dom	domain of function
$\Re$	the set of reals

3.1 Model of Recursive Partitioning

Matrices are typically though of as "flat" objects. We develop an alternative view of matrices as recursively partitioned two-dimensional index spaces called *hierarchical matrix structures*. In order to describe these objects, we define structuring and unstructuring functions. Utilizing these functions we are able to constructively define the set of legal hierarchical matrix structures.

3.1.1 Matrix Index Spaces

The common concept of matrices is that of two dimensional arrays of scalars. More formally they can be viewed as collections of functions that map to values. We will call such matrices, *base matrices*. The following definition describes the set of all finite matrices of real numbers[4].

Def. 3-1 Given $\Re$ *we define M, the set of finite matrices in* $\Re$*, as the set of finite functions from* $N \times N$ *to* $\Re$ *whose domains are the intervals* $\{1..m\} \times \{1..n\}$ *for some natural numbers m and n*

$$M \therefore \left\{ t \in N \times N \xrightarrow[f]{} \Re \mid \exists m,n \text{ such that } dom(t) = \{1..m\} \times \{1..n\} \right\}$$

[4] Similar definitions can be constructed in a straightforward manner to handle matrices with integer or complex values.

For all T in M the row and column sizes are defined as

$$rowsize(T) \therefore \{max\ i \mid \exists j, v \text{ such that } (i, j, v) \in T\}$$

$$colsize(T) \therefore \{max\ j \mid \exists i, v \text{ such that } (i, j, v) \in T\}$$

We generalize the definition of matrices over scalars to the concept of matrices of matrices: recursive matrices. We use the notation S^i to specify the set of recursive matrices that have at most i levels of structuring. The space of recursive matrices can be formulated as follows.

Def. 3-2 The set of recursive matrix spaces, S, is defined inductively as follows.

$$S^1 \therefore M$$

$$S^{i+1} \therefore S^i \cup \left\{ s \in N \times N \xrightarrow[f]{} S^i \mid \exists m, n \text{ such that } dom(s) = \{1..m\} \times \{1..n\} \right\}$$

$$S \therefore \bigcup_i S^i$$

Unfortunately, this definition is not sufficient for our purposes. Although all hierarchical matrices are recursive matrices, not all recursive matrices are legal hierarchical matrices. Therefore, we must specify the restrictions on the set of recursive matrices that exactly define the set of hierarchical matrices. We develop the concept of a *partition* as the basis for these restrictions.

A *partition* is a mapping from a matrix index space to another more structured one. The principal parameters of a partition are the row and column element-size vectors. These vectors describe how the matrix space is to be structured; specifically, the number of elements in each vector determines the new row and column sizes of the resulting structured index space.

Def. 3-3 A row (or column) element-size vector is a one dimensional matrix of positive integers.

$$V \therefore \left\{ v \in N \xrightarrow[f]{} N^+ \mid \exists n \in N \text{ such that } dom(v) = \{1..n\} \right\}$$

The length (number of structured elements) of a vector, R in V, is defined as

$$length(R) \therefore rowsize(R)$$

and the size of the first i elements of the vector is defined by the following recurrence.

$$size(R, 0) \therefore 0$$

$$size(R, i) \therefore R(i) + size(R, i-1)$$

Therefore, the total size of the vector can be defined.

$$size(R) \therefore size(R, length(R))$$

We now define the partition of a matrix as the mapping from a matrix and row and column element-size vectors to a structured matrix.

Def. 3-4 Given a matrix and row and column element-size vectors, the function partition produces a one-level structured matrix ($partition: M \times V \times V \rightarrow S^2$). The domain of the function partition is described by

$$dom(partition) \therefore \{(m,r,c) \in M \times V \times V \mid (rowsize(m) = size(r)) \wedge (colsize(m) = size(c))\}$$

and its definition is given by

$$partition(m,r,c) \therefore s \in S^2 \text{ where } (dom(s) = \{1..length(r)\} \times \{1..length(c)\}) \wedge$$
$$(\forall (i,j) \in dom(s) \text{ such that}$$
$$(dom(s(i,j)) = \{1...r(i)\} \times \{1..c(j)\}) \wedge$$
$$(\forall (k,l) \in dom(s(i,j)) \text{ such that}$$
$$m(size(r,i-1)+k, size(c,j-1)+l) = s(i,j)(k,l))).$$

Since partition is only a partial function, we are interested only in the elements of S^2 to which the partition function actually maps; we will call the *image* of partition in S^2 the set H^2. For every element in H^2 there exists an inverse function for the partition function. This function, *flat*, unpartitions the structured matrix.

Def. 3-5 Given an element h in H^2, the function flat produces the matrix from which it was constructed ($flat: H^2 \Rightarrow M$). Its definition is derived from the function partition as follows.

$$flat(h) \therefore m \in M \text{ where } \exists r,c \text{ such that } partition(m,r,c) = h$$

The concept of partitioning, as currently defined, allows us to generate matrix structures with two levels of structure. Since our goal is to describe matrices that have many levels of structuring, we must generalize the partitioning concept. One alternative is to extend the definition of partitioning so that the function can be applied to elements H^i (structured matrices of depth i); This defines the subset of structured matrices that are obtained through "bottom-up" partitioning. This approach is not entirely desirable because it fails to describe all the legally structured matrices. Therefore, we will explore another approach - that of "top-down" partitioning.

Top-down partitioning is the process of recursively partitioning the elements of a two-level structured matrix (matrices in H^2). We are specifically interested in all the legal top-

down structurings of a given matrix. For a matrix m we will call its set of structurings, H_m, and say that m is the *root* of these matrices.

Def. 3-6 Given a matrix, m in M, the set of legal partitions rooted at that matrix can be defined recursively. The base set is just the matrix m, $H_m^1 \therefore m$.

The second level set can then be defined as union of the base set and the set of legal partitions of m.

$$H_m^2 \therefore H_m^1 \cup \left\{ h \in S^2 \mid \exists r,c \text{ such that } h = partition(m,r,c) \right\}$$

Successive sets are then defined as the union of the previous set and the set described by the addition of another level of partitioning.

$$H_m^{i+1} \therefore H_m^i \cup \Big\{ h \in S^{i+1} \mid \Big(\exists h' \in H_m^2 \text{ such that } \big(dom(h) = dom(h') \big) \wedge \Big(\forall (k,l) \in dom(h) \text{ such that } h(k,l) \in H^i_{h'(k,l)} \Big) \Big) \Big\}$$

And finally we can define the set, H_m, of all legal hierarchical partitions of m.

$$H_m \therefore \bigcup_i H_m^i$$

Using Definition 3-6 we can now define the set of all legal hierarchically structured matrices, H.

Def. 3-7 The set of all legal hierarchical structured matrices, H, is the union of the legal hierarchical structurings of finite matrices.

$$H \therefore \bigcup_{m \in M} H_m$$

3.1.2 Hierarchical Structure Functions

The previous definitions constructed legal hierarchical matrix structures by using the partition function implicitly. As a result, there are almost no explicit relationships among elements of H. The only exceptions are the relations between base matrices and their one-level partitions. We will develop three functions that assert some other useful relations. These functions are special extensions to the partition and flat functions. They are called *r_flat* (for recursive flat), *g_partition* (for generalized partition), and *g_flat* (for generalized flat).

Def. 3-8 The total function r_flat maps hierarchical matrix structures to the their associated base matrices ($r_flat : H \Rightarrow M$*). Its definition is the following.*

$$r_flat(h) \therefore m \in M \text{ where } \exists H_m \subset H \text{ such that } h \in H_m$$

The functions g_partition and g_flat are versions of the partition and flat functions that operate on structured matrices rather than just base matrices.

Def. 3-9 Given a structured matrix and legal row and column partitions, g_partition partitions the structure ($g_partition: H \times V \times V \rightarrow H$). Its domain is described by

$$dom(g_partition) \therefore \{(h,r,c) \in H \times V \times V \mid (rowsize(h) = size(r)) \wedge (colsize(h) = size(c))\}$$

and its definition is given by

$$g_partition(h,r,c) \therefore h' \in H \text{ where } \Big(dom(h') = \{1..length(r)\} \times \{1..length(c)\}\Big) \wedge \Big(\forall (i,j) \in dom(h') \text{ such that } \Big(dom(h'(i,j)) = \{1...r(i)\} \times \{1..c(j)\}\Big) \wedge \Big(\forall (k,l) \in dom(h'(i,j) \text{ such that } \Big(h(size(r,i-1)+k, size(c,j-1)+l) = h'(i,j)(k,l)\Big)\Big)\Big)$$

Def. 3-10 The function g_flat unpartitions one level of structuring of a hierarchically structured matrix ($g_flat: H \rightarrow H \times V \times V$). This function is only applicable to hierarchical matrix structures whose top level structuring results from the use of the function g_partition. Its domain is

$$dom(g_flat) \therefore \{h \in H \mid \exists (h',r,c) \in H \times V \times V \text{ such that g_}partition(h',r,c) = h\}$$

and its definition is derived from the function g_partition as follows.

$$g_flat(h) \therefore (h',r,c) \in H \times V \times V \text{ where } g_partition(h',r,c) = h$$

There are two other useful functions that can be applied to hierarchical matrix structures; we begin with the function, *height*.

Def. 3-11 Given a hierarchical matrix structure, h, we define its height(h) as the smallest i such that h is an element of H^i.

Since our matrices are now "tree-like" rather than single-leveled, we can now apply indexes recursively to obtain elements of the structure. We call such a recursive indexing object an *index sequence*.

Def. 3-12 An index sequence is a mapping from the integers to index tuples.

$$X \therefore \left\{x \in N \xrightarrow[f]{} N^+ \times N^+ \mid \exists n \in N \text{ such that } dom(x) = \{1..n\}\right\}$$

The list functions head and tail return the associated portions of a given sequence in the usual manner. The application of a hierarchical matrix structure, h, to an index sequence, x, is defined as

$$h(\varnothing) \therefore h$$
$$h(x) \therefore h(head(x))(tail(x)).$$

3.1.3 Finite Structures

Now we can focus our attention on the interesting elements of H: those of finite height. The structures of finite height are interesting for reasons of computability. We do not want structures over which primitive recursive functions will have computations of infinite length. We observe that since we begin with only finite sized matrices, infinite height structured matrices can only occur if there exist infinite sub-structure of matrix spaces of size 1-by-1.

Theorem 3-1 A hierarchical matrix structure is of finite height iff it contains no infinite sub-structure of 1-by-1 matrix spaces.

From now on H will only refer to the restricted set of structures of finite height.

3.1.4 Summary

In this section we have developed the set of legal hierarchically structured matrices. In themselves, these structures are of little interest. The purpose they serve is to provide a framework for capturing the special structure of specific matrices. This special structure is described in the form of matrix properties.

3.2 Matrix Properties

Matrix properties capture semantic information about the organization of values in a matrix. Exploitation of this information can lead to more efficient representations and computations. Some of the well known matrix properties include upper triangular, diagonal, tridiagonal, symmetric, identity, and zero. These properties describe some of the kinds of matrices that arise in scientific applications, but they are insufficient for describing the multitude of more complicated matrices that often arise. Hierarchical matrix structures can be used as a framework in which to leverage the standard matrix properties. We define matrix properties as assertions over hierarchical matrix structures. In this form, they provide semantic information to a compiler for performing type checking and for constructing efficient implementations of matrix operations. In addition we define relationships between properties that serve as type coercion transformations.

3.2.1 Hierarchical Matrix Properties

Def. 3-13 A matrix property is an element of the set of functions

$$PROP \therefore M \Rightarrow B$$

that map matrices to the set B of boolean values.

Each element of the set PROP generates a subset of the set M. Clearly it would be impossible to define each of these properties explicitly. Therefore we must choose a very useful subset of them: a subset that includes all the well known matrix properties. Our approach is to define the set of matrix properties that can be composed over legal hierarchical matrix structures.

The key idea is to determine hierarchical definitions for the well-known matrix properties. The definitions are constructed recursively so that for structures they express structure constraints and for base matrices the definitions resolve to the base definitions (as defined in standard matrix algebra texts [13]). For example, the definition of the property zero for a base matrix states that the value zero is returned for all indices. In the hierarchical definition, the associated statement is that all indices return zero structures. We are interested in describing these structural assertions as exemplified by the following definition for zero structures.

Def. 3-14 The zero property for a matrix structure h in H.

$$zero(h) \therefore \forall (i,j) \in dom(h) \text{ where } zero(h(i,j))$$

There exist a number other properties whose definitions are based on the existence and location of zero structures. Some representative examples include diagonal, lower bidiagonal, unit upper bidiagonal, tridiagonal, upper hessenberg, and upper triangular. The following a few representative definitions.

Def. 3-15 The lower bidiagonal property imposes the following constraint on matrix structures h in H.

$$lower_bidiagonal(h) \therefore \forall (i,j) \in dom(h) \text{ where } \left((j > i) \vee (i > j+1)\right) \text{ implies } zero(h(i,j))$$

Def. 3-16 The tridiagonal property imposes the following constraint on matrix structures h in H.

$$tridiagonal(h) \therefore \forall (i,j) \in dom(h) \text{ where } \left((j > i+1) \vee (i > j+1)\right) \text{ implies } zero(h(i,j))$$

Def. 3-17 The upper triangular property imposes the following constraint on matrix structures h in H.

$$upper_triangular(h) \therefore \forall (i,j) \in dom(h) \text{ where } (i > j) \text{ implies } zero(h(i,j))$$

3.2.2 Strong Recursive Properties

These last three definitions bring up an interesting point about the nature of structure: what is the relationship between the property of a structured matrix and that of a base matrix? For example, is an upper triangular structure necessarily an upper triangular base matrix? Since the definitions are assertions over structure, and not value, this relationship is not guaranteed to hold. If we examine the definition of the zero property we notice that it is expressed in such a way that if the property zero holds over a structure then it also holds over its root matrix. We call such definitions, strong recursive definitions and write their names with overbars.

Def. 3-18 A property $\overline{prop}$ is strong recursive if the following statement holds.

$$strong_recursive(\overline{prop}) \therefore \forall h \exists m \text{ such that}$$
$$h \in H_m \wedge (\overline{prop}(h) \text{ implies } \overline{prop}(m))$$

It is sometimes necessary to define helper functions in order to define strong recursive properties. An example of a helper function is the function square. It states that the row dimensions and column dimensions of a structure are equivalent.

Def. 3-19 A matrix structure h in H is square iff its row dimension equals its column dimension.

With the addition of the helper function square, it is possible to define a number of strong recursive properties including upper triangular, lower triangular, unit upper triangular, unit lower triangular, and symmetric. The following definition for the strong recursive upper triangular property is a representative example.

Def. 3-20 The strong recursive upper triangular property for a matrix structure h in H is defined as follows.

$$\overline{upper_triangular}(h) \therefore square(h) \wedge$$
$$\left(\forall (i,j) \in dom(h) \text{ where}\right.$$
$$\left.\left((i > j) \text{ implies } zero\left(h(i,j)\right)\right) \wedge \left((i = j) \text{ implies } \overline{upper_triangular}(h(i,j))\right)\right)$$

It is additionally possible with the use of other helper functions to construct strong recursive definitions for properties such as tridiagonal and positive definite.

3.2.3 Representations

One major benefit of matrix types is the possibility of having special matrix representations. Representations are space efficient methods for describing the structure and values of a matrix in the memory of a computer. They encode the matrix information and provide mappings for obtaining needed values. Since matrices require storage space that is the square in their size, special representations can result in great savings.

Consider a matrix with the zero property. As a direct result of this information, no specific value need to be stored. It can be directly computed that all indices that return scalars will return the value 0. Clearly such a scheme has the advantage of yielding a constant space representation.

Properties that specify the location of zero elements are not the only kind of matrix properties that can be exploited for space efficient representations; there are a number of matrix properties that describe different kinds of value replication. Examples of this latter class of properties includes symmetry, Hankel, and Toeplitz. Symmetric matrices can be stored in half the space that is required for the whole matrix. Hankel matrices, since they are symmetric in a different direction, share this same representational characteristic. Toeplitz matrices have even greater savings because they can be stored in space that is linear in the size of the matrix. Combinations of the properties, such as symmetric Toeplitz, yield the combined savings of their constituent properties.

3.2.4 Types and Functions

The importance of strong recursive properties is that they demonstrate a relationship between matrix properties and structuring functions. It is useful to develop as many of these relationships as possible because they provide the basis for automatic structure transformations by a compiler. Therefore, we will develop a number of other relationships and demonstrate how they preserve property information even after structuring operations. We will begin with some relationships between strong recursive properties and structuring functions.

Theorem 3-2 Given a matrix structure h in H and some strong recursive property $\overline{prop}$ *such that* $\overline{prop}(h)$ *holds then* $\overline{prop}(r_flat(h))$*also holds.*

There is also the obvious corollary to this theorem.

Cor. 3-1 Given a matrix structure h in H^2 *and some strong recursive property* $\overline{prop}$ *such that* $\overline{prop}(h)$ *holds then* $\overline{prop}(flat(h))$ *also holds.*

The previous theorems have exploited characteristics of the flat functions. There also exist relationships between partition functions and the matrix properties. Consider the partition of an upper triangular base matrix. For most legal row and column partitions the resulting

structured matrix is neither strong recursive upper triangular nor upper triangular. But clearly there do exist some partitionings that propagate the property.

One useful kind of partition is an *equipartition* where the row and column vectors of element sizes are equal. Using equipartitions it is possible to state a number of partitioning theorems. The following is an example theorem for upper triangular matrices.

Theorem 3-3 Given a matrix structure h in H for which upper_triangular(h) holds. For any legal equipartitioning, h´ = g_partition(h,r,r), upper_triangular(h´) holds.

3.2.5 Type Coercion

In the previous section we examined some of the relationships between typed matrices through the use of structuring functions. We now explore another set of relationships that are not created via structure, but rather by the nature of the property definitions themselves. When we examine some of the property definitions carefully we notice that some of them actually subsume others of them. For example, the definition for $\overline{\text{upper_triangular}}$ subsumes the definition for upper_triangular.

Theorem 3-4 For all h in H if $\overline{\textit{upper_triangular}}$*(h) holds then upper_triangular(h) also holds.*

Another example of a subsumption relationship is that between lower_bidiagonal and tridiagonal.

Theorem 3-5 For all h in H if lower_bidiagonal(h) holds then tridiagonal(h) also holds.

A number of other relationships exist. For example, another one that utilizes the definitions presented in this chapter is the subsumption relationship between tridiagonal and upper_Hessenberg.

3.3 Operations and Algorithms

Matrix operations are functions that map input matrices to output matrices. These functions are not total but rather are defined over *compatible* matrices; for example, matrices are compatible for addition if they have matching row sizes and column sizes. Also, there are many possible function implementations which we will call *algorithms*. Many of the algorithms are type-specific. That is, they are only correct if applied to input matrices of certain types. The importance of type-specific algorithms is that by exploiting type information they can perform an operation with less computation. We will now develop a framework for expressing implementations of hierarchical matrix operations. This framework is specifically designed to support the construction of efficient type-specific implementation of matrix operations by a compiler.

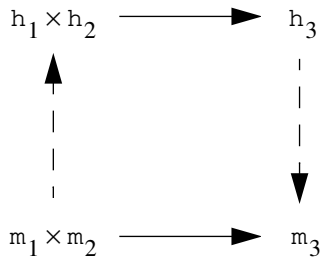


Figure 3-1. A commuting diagram for a binary matrix operation. The matrices of type m_i are matrices of scalars and those of type h_i are hierarchical matrices. The dashed arrows represent structuring and unstructuring operations and the solid represent operation application.

3.3.1 Hierarchical Matrix Operations

Hierarchical matrix operations are extensions of the standard matrix operations. Figure 3-1 shows one view of a hierarchical matrix operation. In words, the picture depicts the rule that for a given matrix operation and compatible input base matrices there exists hierarchical structurings of the base matrices such that the application of the hierarchical matrix operation over the structured matrices results in a structured matrix which is a hierarchical structuring of the result of the matrix operation over the base matrices.

If we generalize this picture we obtain a collection of graphs such as the one shown in Figure 3-1. Each of these graphs is anchored at a pair of compatible typed base matrices and represents all the legal hierarchical structurings that are rooted at those matrices. Our goal is to determine implementations for the matrix operations over structured matrices. We can consider the implementation of the function for the typed base matrices a solved problem. Many matrix packages [1][5] provide implementations of these operations. In addition, the algebraic considerations of determining both the type and the size of the result matrix from the sizes and types of the inputs are well understood. For the structured versions of the matrices the implementation of these functions is not as straightforward.

There are two primary issues that must be addressed for structured implementations. First, any implementation of the function must handle hierarchical type descriptions. It apparently could be as specific as one version of the function for structured input matrices. This issue of specificity is reminiscent of the problem of types for matrices. Second, we must determine the relationship between structured input matrices and structured output matrices. The principal constraint is that the structured result matrix must be a hierarchical structuring of the base result matrix. We solve both of these problems by determining a recursive implementation approach that is isomorphic to the approach taken for hierarchical matrix types.

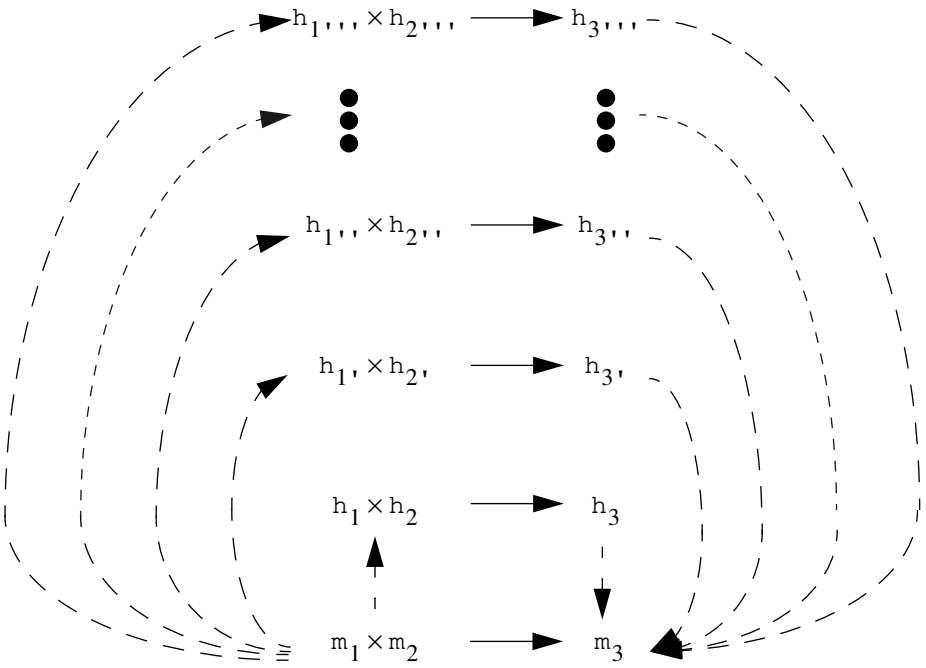


Figure 3-2. The full commuting diagram for a binary matrix operation. The h matrices are all the legal hierarchical structurings of the root matrices m.

3.3.2 Compatibility Constraints

The key to a recursive implementation approach is a recursive formulation of compatibility. For base matrices, compatibility is defined in terms of row and column sizes. For example, the compatibility constraint for matrix addition is given by the following definition.

Def. 3-21 For the expression A+B=C, add_compatible is given by

$$add_compatible(A,B,C) \therefore \left(rowsize(A) = rowsize(B) = rowsize(C)\right) \wedge \left(colsize(A) = colsize(B) = colsize(C)\right).$$

Similarly, the base matrix compatibility constraint for multiplication is given by the following definition.

*Def. 3-22 For the expression A*B=C, mult_compatible is given by*

$$mult_compatible(A,B,C) \therefore \left(colsize(A) = rowsize(B)\right) \wedge \left(rowsize(A) = rowsize(C)\right) \wedge \left(colsize(B) = colsize(C)\right).$$

We will now use some generalized definitions of matrix addition and matrix multiplication for developing recursive versions of these compatibility conditions. The first generalized definition is for matrix addition.

Def. 3-23 The operation matrix addition for the expression A+B=C with A,B, and C in M, is defined by the following constraint.

$$+_M(A,B,C) \therefore \forall (i,j) \in dom(C) \text{ where}$$
$$add_compatible(A,B,C) \ \wedge \ \big(A(i,j) + B(i,j) = C(i,j)\big)$$

Since our goal is to develop recursive implementations of matrix operations, we will now consider the possibility of A, B, and C not being elements of M. In other words, is Definition 3-23 still meaningful if A, B, and C are structures? By applying this definition recursively for the sub-operation of A(i,j) + B(i,j) = C(i,j) we obtain meaningful structure and value constraints. This statement is captured by the following theorem.

Theorem 3-6 For the expression A+B=C with A,B, and C in H, if Definition 3-23 holds recursively then that operation is equivalent to matrix addition.

We can now take the same approach to defining the constraint for a recursive matrix multiplication.

*Def. 3-24 The operation matrix multiplication for the expression A*B=C with A,B, and C in M, is defined by the following constraint.*

$$*_M(A,B,C) \therefore \forall (i,j) \in dom(C) \text{ where}$$
$$mult_compatible(A,B,C) \ \wedge \left(C(i,j) = \sum_{k=1}^{colsize(A)} A(i,k) * B(k,j) \right)$$

The next step is to construct a recursive version of this definition.

*Theorem 3-7 For the expression A*B=C with A,B, and C in H, if Definition 3-24 holds recursively then that operation is equivalent to matrix multiplication.*

One interesting feature of Theorem 3-7 is that it not only expresses an explicit structural constraint among the matrices but it also expresses some implicit structural constraints within the matrices. These implicit constraints arise in Definition 3-24 from two sources: multiple pairs of elements are multiplied together and the groups of results of multiplications are summed. The former constrains both elements in each column of A to have equivalent numbers of column elements and elements in each row of B to have equivalent numbers of row elements; this constraint holds recursively over both these types of elements. The latter constrains the multiplications within an inner product to result in addition compatible matrices. This constraint in turn restricts both rows of A to have row equivalent elements and columns of B to have column equivalent elements.

Using this same approach we can obtain recursive constraints for a number of other matrix operations. The resulting constraints will limit the graphs, such as in Figure 3-1, to recursively compatible hierarchically structured matrices.

3.3.3 Algorithms

These constraints provide a framework, not a solution to our problem of determining efficient algorithms for matrix operations over hierarchical matrix structures. The primary problem is that they make no mention of matrix properties. One could imagine constructing special recursive algorithms for matrices that have strong recursive matrix properties[5]. But this would be inadequate for the considerably greater number of otherwise typed hierarchical matrix structures.

Our approach will be to construct composable, type-specific, abstract algorithms. They are in a sense "near-sighted" algorithms in that do not use index sequences, only indices. So they can make no assumptions about the structure of what is returned by an index function other than that its either a base matrix or a a matrix structure with known row and column sizes. Therefore, the operations that comprise these algorithms are abstract operations that can either be algorithms with the same abstraction characteristic, or base matrix algorithms. The question is how do we develop algorithms with this special property?

Fortunately, there does exist a large collection of algorithms that are called "block algorithms." They derive their name from the fact that they treat matrices as being composed of contiguous, rectangular subregions. The block algorithms operate on these sub-regions forming sub-results. It seems quite reasonable that through some kind of abstraction that we could turn these block algorithms into suitable abstract algorithms. But how would we know that these algorithms would still be correct over arbitrary compatible matrix structures?

The claim is that a block algorithm will operate correctly over any compatible matrix structures if it is possible to convert all its calls to specific base algorithms into calls to abstract matrix operations. The idea is that we could implement each of these matrix operations with some recursive algorithm. We will call a block algorithm that has this property an abstract block algorithm or algorithm for short.

Def. 3-25 We define an abstract block algorithm as a legal block algorithm in which all constituent matrix sub-operations are in terms of recursively defined matrix operations.

These algorithms serve as composable elements for constructing type-specific recursive implementations for matrix operations.

Theorem 3-8 Abstract block algorithms are hierarchically composable over compatible matrix structures

---

[5] This is in effect the approach of Skagestein discussed in Chapter 2.

As a result, for a given operation, we obtain a one-to-one mapping between compatible input matrix structures, an implementation, and the output structures. Not only can we determine the output structure, but we can also determine the output type.

3.3.4 Type Propagation

Type propagation is the term used for describing the determination of output type from input type. For example, for the matrix multiplication operation, the input types lower triangular and lower triangular yield lower triangular as the result type. Clearly it would be tedious to compute all the output types given all the legal input types. Since we are composing implementations, we can equally well compose the result type in the same manner. Therefore, we can view type propagation as a level by level process.

3.4 Metrics and Optimization

In the previous three sections we have explored a theory for describing typed hierarchical matrix structures and a framework for composing type-specific algorithms as implementations of matrix operations. In this section we will discuss some issues in constructing an implementation of a compiler for operations over hierarchically structured matrices.

3.4.1 Metric Space

The task of the compiler is to generate efficient implementations that exploit the semantic information in the hierarchical matrix structures. For the sake of efficiency or implementability it may be necessary for a compiler to alter the structure of the hierarchical matrix structures in a computation. We will consider the issue of efficiency first.

Up to this point we have implicitly considered type-specificity to be the only issue of concern. But on a real machine we will have to consider other metrics in order to obtain efficient implementations. On modern hierarchical memory computers the principal concerns have to do with data movement; these machines have multiple levels of memory where in general the faster the memory the smaller its size. For the sake of efficiency it is desirable to perform as much of the computation as possible within the fastest levels of memory; one important metric for measuring this is cache utilization.

This model of performance suggests the strategy of analyzing the access patterns and granularity of an algorithm in order to determine its runtime. The resulting parameterized equations can be used for labelling the algorithms with associated values. This labelling is useful for evaluating the efficiency of an algorithm applied to compatible hierarchical matrix structures. But how do we deal with incompatible structures? Also, what should we do if the needed algorithm is not coded in the compiler?

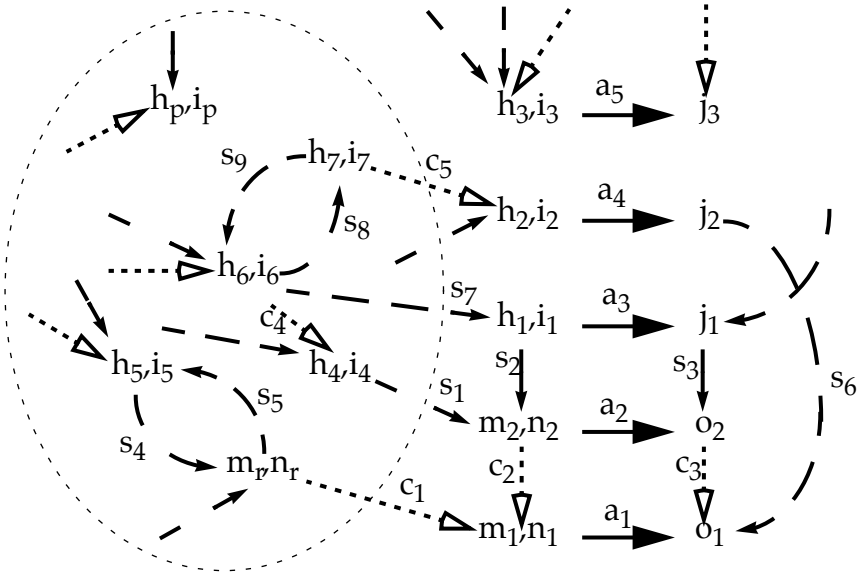


Figure 3-3. A matrix operation graph. This graph represents some matrix structures and functions associated with matrix operation: the letters m,n, and o are used for base matrices, the letters h,i, and j are used for structured matrices, the letter a is used for values labelling algorithms, the letter s is used for labelling structuring and unstructuring functions, and the letter c is used for type coercion functions.

3.4.2 Optimization Graph

The solution to both these problems is to utilize the mapping functions as a means to move among legal hierarchical matrix structures. This suggests viewing the space of structures and functions as a graph. If we construct data movement models, the resulting values can be used for labelling the edges of these graphs. An example graph of this kind is shown in Figure 3-3.

In this graph, the nodes are pairs of input matrices or an output matrix in a binary operation and the edges represent different kinds of mapping functions. The labellings on the edges denote costs under a certain set of metrics. Such a graph gives a very interesting snapshot of the capabilities of a compiler. We will briefly examine some of these features.

One important feature of a compiler is “implementability.” That is, characteristics of how the compiler implements an operation. In our example graph the dense arrows represent matrix algorithms. Therefore, the pairs of matrices that have a dense arrow are ones that are implemented in the compiler; the pairs m_1,n_1 and h_1,i_1 are examples of ones that are directly implemented. Conversely, the pairs m_r,n_r and h_6,i_6 are not directly implemented. But both of these pairs may be implemented through a number of function applications. This characteris-

tic is not universally true in the graph; for example, the pair h_p,i_p has no implementation whatsoever in the current graph.

In the context of these compiler graphs, implementation optimization can be viewed as the process of determining a path[6] of least cost from the input matrices to some output matrix. Since a computation may involve a number of matrix operations that share input or output matrices, the optimization process is not really this straightforward. A more realistic goal is to minimize the total cost of the mapping functions throughout the whole computation. In fact, these paths are hierarchical. Since each algorithm is composed of sub-operations, the matrix structures involved in these operations may themselves undergo some mapping for the sake of optimization.

3.5 Conclusions

In this chapter we have developed a formal theory for matrix computations that allows the inclusion of semantic information. This theory describes the concept of hierarchically structured matrices that are annotated with matrix properties; these structures are constructed by the application of structuring (and unstructuring) functions. Additionally we explored some of the relationships among legal hierarchical matrix structures.

Next we addressed the issue of matrix operations over these structures. The idea of recursive compatibility allowed us to determine a one-to-one mapping between typed input matrices and typed output matrices. This mapping defined an implementation for a matrix operation. In order to handle the multitude of matrix properties we composed the implementations out of abstract algorithms derived from well-known block algorithms. These algorithms had the additional benefit of allowing us to determine not only the result structure but also the result type.

This approach of hierarchical composition provided the basis for an implementation strategy. The idea was to view the matrix structures and the mapping functions as the nodes and edges of a labelled graph. In this graph, annotated paths from input matrix structures to output matrix structures represent implementations and their expected runtimes. Therefore, compiler optimizations for finding efficient implementations become equivalent to determining minimum cost paths in these graphs.

---

[6] As we have seen above, sometimes no path exists.

Chapter 4

MaTRiX+$^+$ Language

The MaTRiX+$^+$ language provides a C++ interface for constructing structured matrix computations. Specifically, the language mechanisms supply a declarative means of specifying hierarchical matrix structures for describing matrices that arise in modern scientific applications, data distributions of these matrices across distributed memory architectures, and the matrix operations to be performed over them. The interface also hides unnecessary representation and implementation detail from the user. For example, MaTRiX+$^+$ separates the construction of structures, which are types, and the generation of instances, which are specific matrix objects that will contain values. In addition, the user's only access to parallel execution characteristics is specification of data distributions and the description of a hierarchical mesh of virtual processors. Also, the actual matrix operations are accessed through simple interfaces that mask all implementation detail.

This chapter describes the syntax of the principal elements of the MaTRiX+$^+$ language. We begin with a brief introduction to object-oriented programming in order to clarify the style of the MaTRiX+$^+$ language interface. The description of the language begins with the presentation of matrix types and their associated functions. Next, we describe the constructs for specifying a hierarchical mesh of processors. Then, the syntax of matrix objects and their relationship to matrix types and processor meshes is discussed. Examples of code are presented in Chapter 5.

4.1 Object-Oriented Interface

C++ [9] is an object-oriented extension to the C language. The key concepts that it adds that concern the MaTRiX+$^+$ language interface are those of *classes* and *objects*. A class

is an abstract data type: a collection of data encapsulated with their associated functions. In C++ a class is treated like a type (integer for example). A object is an instance of a class. Objects are created by a special class function called a constructor. This function can be parameterized in order to be able to construct objects with different initial data.

Since the MaTRiX+$^+$ prototype is a collection of C++ classes, the user is primarily constructing objects in his specifications[7]. The style of the MaTRiX+$^+$ language interface reflects this fact.

4.2 Hierarchical Matrix Types

MaTRiX+$^+$ provides language mechanisms for defining hierarchical matrix types. These mechanisms are used to describe typed matrices and to compose them into hierarchical structures that are typechecked for legality. In addition, data distribution directives can be optionally associated with a matrix type. The full syntax for declaring a named typed matrix is as follows (keywords in the MaTRiX+$^+$ language are in bold and have the prefix MTRX in order to avoid name collisions with other packages):

MTRX_Type type_name(rowdim, coldim, Property, BaseType, Distribution);

The optional fields for a matrix type declaration are the property descriptor, the base type descriptor, and the data distribution directive; it is illegal to give both a base type and a distribution since base matrices cannot be distributed.

The syntax for a property descriptor is as follows.

MTRX_Property prop_name($prop_1$,$prop_2$, ..., $prop_n$);

The purpose of the property descriptor syntax is to simplify the description of the matrix property; it obviates the need for the user to memorize long type names. Instead the user composes the necessary base properties in an order independent manner. These properties are chosen from Table 4-1 such that at most one property can be chosen from each category. So for example, a matrix which is symmetric, tridiagonal, and positive definite can be declared in the following way.

MTRX_Property my_type(**MTRX_Symm**, **MTRX_Tridiag**, **MTRX_Pos_Def**);

The base type field is used for declaring the data type of a base matrix. The legal choices for a base matrix are: **MTRX_INT**, **MTRX_FLOAT**, **MTRX_DOUBLE**, and **MTRX_COMPLEX**. In the case that no base type is given, a hierarchical matrix type is by default defined[8].

---

[7] In fact the MaTRiX+$^+$ system is written in CC++ classes, but the difference is not important here.

[8] The exceptions to this rule are MTRX_Zero and MTRX_Identity matrices. They are both base matrix types and hierarchical matrix types.

For hierarchical matrix types it is legal to have a data distribution directive. A directive tells how the elements of the instantiated hierarchical matrix type are to be distributed relative to a local description of the processor mesh[9]. That is, the distribution for a matrix type is not absolute, but rather relative to the context in which the matrix type is instantiated. The MaTRiX+$^+$ system supports three kinds of distributions named **MTRX_BLOCK**, **MTRX_CYCLIC**, and **MTRX_ALL**. The semantics of these directives will be discussed in Section 4.4. The default distribution, if no directive is specified, is **MTRX_CYCLIC**. This promotes the exploitation of parallelism whenever possible.

Table 4-1. The base matrix properties

Category	Properties
Banded	**MTRX_Diag**, **MTRX_Upper_bidiag**, **MTRX_Lower_bidiag**, **MTRX_Tridiag**, **MTRX_Pentadiag**, **MTRX_Upper_triang**, **MTRX_Lower_triang**, **MTRX_Upper_Hessen**, **MTRX_Lower_Hessen**
Symmetric	**MTRX_Symm**, **MTRX_Skew_symm**
Hankel	**MTRX_Hankel**
Toeplitz	**MTRX_Toeplitz**
Algebraic	**MTRX_Zero**, **MTRX_Identity**
Numeric	**MTRX_S_diag_dominant**, **MTRX_Pos_definite**
Unit	**MTRX_Unit**
Arrow	**MTRX_Arrow**

For hierarchical matrix types, it is additionally necessary to set the types of their elements. MaTRiX+$^+$ provides methods for selecting the region of a structure that is to be set to a given type. These methods include element, row, column, submatrix, and diagonal. The following describes the syntaxes of these selector objects[10].

MTRX_Element selector_name(row_{int}, col_{int});

MTRX_Row selector_name(row_{int});

MTRX_Row selector_name($row1_{int}$, $row2_{int}$);

---

[9] We will discuss processor meshes in the Section 4.3.

[10] Matrix numbering begins at row 0 and column 0. Also, the main diagonal of a matrix is number 0 with diagonals above numbered with increasing positive integers and diagonals below numbered with decreasing negative integers.

MTRX_Col selector_name(col_{int});

MTRX_Col selector_name($col1_{int}$, $col2_{int}$);

MTRX_Submatrix selector_name($row1_{int}$, $col1_{int}$, $row2_{int}$, $col2_{int}$);

MTRX_Diag selector_name($diag_{int}$);

MTRX_Diag selector_name($diag1_{int}$,$diag2_{int}$);

The type setting function[11] assigns the same matrix type to all elements of a given selected region.

$type_name_1$.**set_type**(region_selector, $type_name_2$);

As an example for using the MaTRiX+$^+$ syntax, the following program fragment declares some of the information about the structure of our example matrix in Figure 1-2 on page 5.

MTRX_Type Submarine(2,2);

MTRX_Property tridiag(**MTRX_Tridiag**);

MTRX_Type RegionA(100,100,tridiag);

MTRX_Type BaseBlocks(10,10,**MTRX_FLOAT**);

MTRX_Diag diag;

MTRX_Element element;

RegionA.**set_type**(diag(-1,1),BaseBlocks);

Submarine.**set_type**(element(0,0),RegionA);

Given the full description of the type of a matrix structure, it is possible to generate a representation of the matrix and to perform operations over it. But first we will look at the mechanism for describing the architecture upon which the matrices will be represented.

4.3 Processor Meshes

The processor mesh descriptor captures a hierarchical view of the processor resources of the architecture. This mesh description, in conjunction with the distribution directives, provides the compiler with the necessary information for partitioning both the matrix objects and

[11] One thing to note about the semantics of this function is that once a matrix type is used to define a region of another matrix type it may no longer be changed.

the computation over the processors in a hierarchical manner. The user's task in the MaTRiX+$^+$ language is to specify a single bottom up partitioning of the processor mesh for the given program. The name of this processor mesh is **MTRX_Proc_Mesh**. It is initialized by the call **MTRX_Proc_Mesh**(p,q). A mesh, once initialized, can be partitioned by the function call **MTRX_Proc_Mesh**.**partition**(m,n) where m < p and n < q. The result is a new top level description of the processor mesh which is of size m by n where each element of the mesh is a sub-mesh of size roughly[12] p/m by q/n. Partitioning may be repeated until the mesh size is 1 by 1. An example hierarchical processor mesh is shown in Figure 4-1.

Given a partitioned mesh, there are a number of functions that provide access to its descriptor. The function ψ is the name for the mesh descriptor in the current context. Its size is given by the functions ψ_{row} and ψ_{col}. Descriptors for sub-meshes are accessed by indexing; the index function $\psi[i,j]$ is meaningful if $1 \le i \le \psi_{row}$ and $1 \le j \le \psi_{col}$. An example of a processor mesh index function is shown in Figure 4-1. It is also possible to access the underlying names represented by the mesh descriptor in the current context through the function ψ. A processor $P_{i,j}$ is said to be in a descriptor if $P_{i,j} \in \psi$.

4.4 Hierarchical Matrix Objects

A hierarchical matrix object is an instance of a hierarchical matrix type. Unlike the type description, an object has a specific representation and can hold data values. The MaTRiX+$^+$ language interface provides methods for defining matrix objects from matrix types and for inputting and outputting data values.

4.4.1 Matrix Object Instantiation

Matrix objects are defined from matrix types through a simple syntax:

MTRX_Object object_name(type_name);

The resulting instantiation of the matrix object is not as straightforward. It is based on matrix property information, the distribution type, and the processor mesh descriptor. The matrix property information can yield a number of different kinds of representations. Additionally, these representations depend on whether the matrix structure is hierarchical or base. Table 4-2 gives the representation classes for hierarchical matrix structures. So, for example, a hierarchical matrix structure with the property **MTRX_Toeplitz** can be represented through the first element of each diagonal. Therefore, an n by n Toeplitz matrix requires only 2*n-1 space. It is also possible for a matrix structure to have a number of representational characteristics. If our Toeplitz matrix were symmetric then it would only require n elements to be represented. Furthermore, if it were also tridiagonal, only two elements would be needed.

---

[12] The partitioning function is the same as the MTRX_BLOCK distribution type that will be described in detail in Section 4.4.

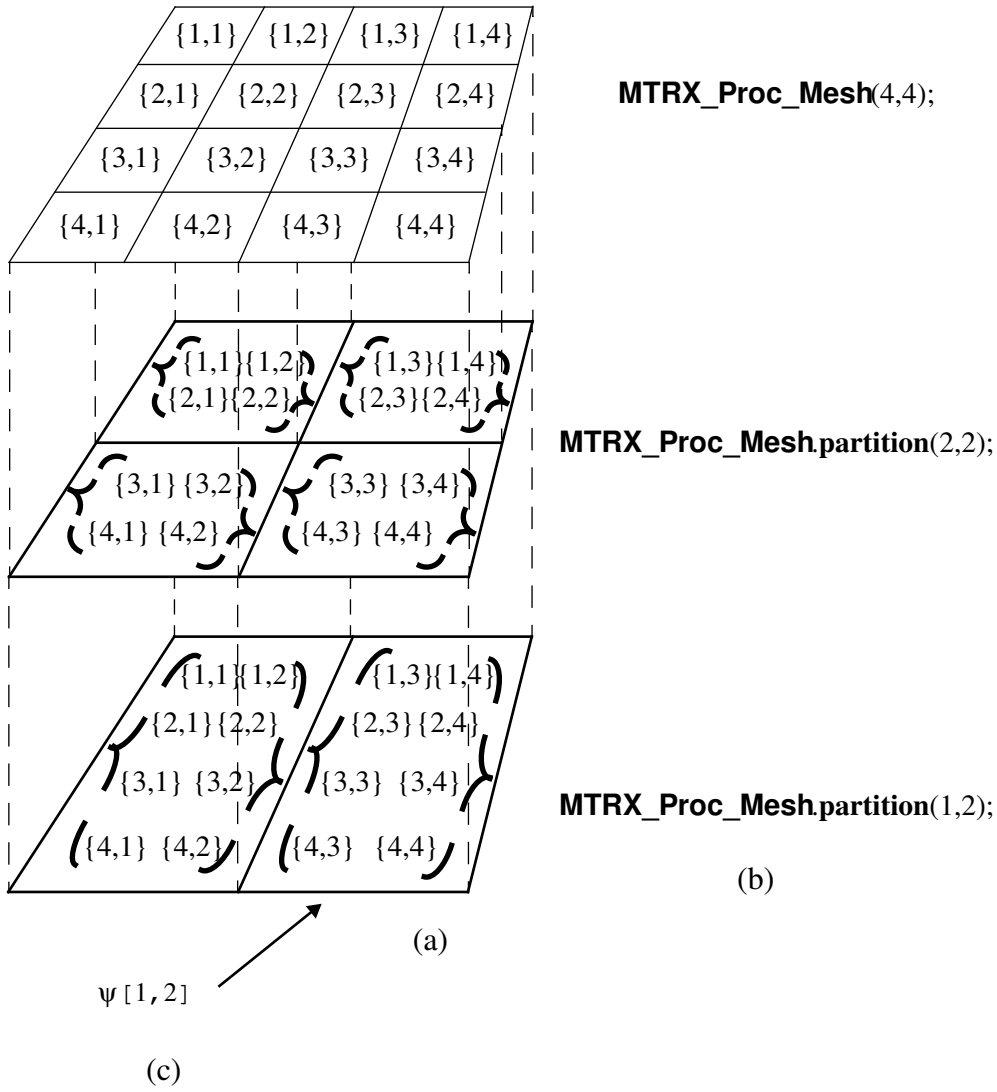


Figure 4-1. The hierarchical processor mesh: (a) on the left is created by the associated MaTRiX+$^+$ functions (b) on the right. Also, a processor mesh index function is shown in (c).

The other factor that affects matrix structure representation is the data distribution type and the processor mesh descriptor. For each element of a hierarchical matrix structure the distribution can be **MTRX_BLOCK**, **MTRX_CYCLIC**, or **MTRX_ALL**. The first two distributions are analogous to the commonly used BLOCK and CYCLIC; our definitions are adapted from those presented in [10]. They differ in that they are defined over hierarchical processor mesh descriptors and are “fairer” distributions as we will discuss later.

The directive **MTRX_BLOCK** distributes the elements of a hierarchical matrix object over the processor mesh in the same way as a partition. Typically for block distributions,

Table 4-2. Representation categories

Representation Class	Matrix Property	Default Representation
Constant	**MTRX_Zero**	size characteristics
	MTRX_Identity	size characteristics
Triangular	**MTRX_Symm**	lower-left triangle
	MTRX_Skew_symm	lower-left triangle
	MTRX_Hankel	upper-left triangle
Vector	**MTRX_Toeplitz**	band headers
Banded	**MTRX_Diag**	necessary bands
	MTRX_Upper_bidiag	
	MTRX_Lower_bidiag	
	MTRX_Tridiag	
	MTRX_Pentadiag	
	MTRX_Upper_triang	
	MTRX_Lower_triang	
	MTRX_Upper_Hessen	
	MTRX_Lower_Hessen	

given an m-by-n matrix object, A, and processor mesh descriptor, ψ, element A[i,j] of the matrix object would be annotated with following element of the mesh descriptor.

$$\psi\left[\left\lfloor \frac{i}{\lceil m/\psi_{row} \rceil} \right\rfloor, \left\lfloor \frac{j}{\lceil n/\psi_{col} \rceil} \right\rfloor\right]$$

The problem with this distribution is that the last processor of a row or column may receive far fewer elements than the other processors. In fact, it is possible for a processor to receive up to $\lceil m/\psi_{row} \rceil - 1$ fewer rows of elements and $\lceil n/\psi_{col} \rceil - 1$ columns of elements than another processor. A fairer distribution is one that is analogous to a round-robin scheme. This method is equivalent to the usual block partition in the case of the number of elements being

divisible by the mesh size. Therefore, we will write the **MTRX_BLOCK** distribution as an element annotation of the form

$$\psi\left[\left\lfloor \frac{\mathtt{i}}{\text{round_robin}\left(\mathtt{m}, \psi_{\mathtt{row}}, \mathtt{i}\right)}\right\rfloor, \left\lfloor \frac{\mathtt{j}}{\text{round_robin}\left(\mathtt{n}, \psi_{\mathtt{col}}, \mathtt{j}\right)}\right\rfloor\right]$$

where the function round_robin is defined as follows.

round_robin(mtrx_size,mesh_size,elt_num) $\therefore$

if $\quad \text{elt_num} < (\mathtt{n}\%\text{mesh_size}) \cdot \left\lceil \frac{\mathtt{n}}{\text{mesh_size}} \right\rceil - 1$

then $\quad \left\lceil \frac{\mathtt{n}}{\text{mesh_size}} \right\rceil$

else $\quad \frac{\mathtt{n}}{\text{mesh_size}}$

This definition of MTRX_BLOCK has the property that the processors will differ in at most one in the number of rows or columns of elements. The annotation equations for the other two distribution directives are more straightforward.

The directive MTRX_CYCLIC performs the well known two-dimensional cyclic distribution. Given an m by n matrix object A and ψ, element A[i,j] of the matrix object is annotated with element of the mesh descriptor.

$$\psi\left[\mathtt{i} \bmod \psi_{\mathtt{row}}, \mathtt{j} \bmod \psi_{\mathtt{col}}\right]$$

The distribution MTRX_ALL specifies that the elements of the matrix type are left undistributed over the processor mesh. Each element of the matrix object is annotated with the entire local processor mesh descriptor. Therefore, A[i,j] is annotated with ψ itself. The directive MTRX_ALL implies replication of a matrix object rather then its distribution. Figure 4-2 shows the different distributions of a 4-by-4 matrix over a 2-by-2 processor mesh.

4.4.2 Access Functions

Once initialized, a matrix object can input and output values and have matrix operations performed upon it. The following are the syntaxes for value functions.

matrix_object(row_{int},col_{int}) = value;

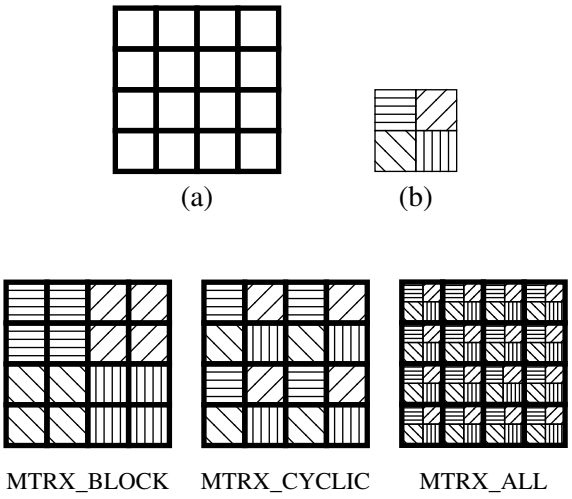


Figure 4-2. Distributions of a 4-by-4 matrix (a) over a 2-by-2 processor mesh (b).

value = matrix_object(row_{int},col_{int});

The next section describes the application and nature of the matrix operation syntax.

4.5 Matrix Operations

The current MaTRiX+$^{+}$ prototype provides the set of matrix operations necessary for solving linear systems of equations. These operations include addition, multiplication, solve, LU decomposition, LL decomposition (Cholesky), forward substitution, and backwards substitution. The syntax for these matrix operations over matrix objects is as follows[13].

Matrix addition has two different forms: A+B→C

MTRX_Add(A,B,C);

and A-B→C

MTRX_Sub(A,B,C);

Matrix multiplication has the usual binary form A*B→C

MTRX_Mult(A,B,C);

and also the general multiplication forms A*B+C→D

MTRX_Mult(A,B,C,D);

and the scaled for αA*B+βC→D

[13] The syntax for the matrix operations are similar in spirit to the interfaces for the LAPACK and BLAS routines.

MTRX_Mult(A,B,C,D,α,β);

There are a number of matrix operations involved with the solving of systems of equations. Given the equation A*x=b, the following operation solves for x directly.

MTRX_Solve(A,b,x);

It is also possible to perform the suboperations that comprise a matrix solve. A matrix decomposition factors a matrix into lower and upper triangles such that A=L*U. MaTRiX++ provides a decomposition object called a **MTRX_Factor** whose constituent triangles can be accessed with the functions **lower** and **upper**. The following is the generic interface for matrix decompositions (A→F).

MTRX_Decomp(A,F)

There also exist interfaces for the specific LU and LL (Cholesky) decompositions.

MTRX_LU(A,F);

MTRX_LL(A,F);

Given an upper or lower triangular matrix[14] it is possible to do a solve via either forwards or backwards substitution. When solving for x in the equations L*x=b or x*U=b the same operation interface suffices.

MTRX_Triang_Solve(L,b,x);

MTRX_Triang_Solve(b,U,x);

4.6 Summary

The MaTRiX+[+] supplies a small number of constructs for the declarative specification of hierarchical matrix structures and their distributions over the separate memories of a collection of processors. In addition, the language provides simple interfaces for well-known matrix operations. The implementation of the distributed matrix objects and operations will be described in the next chapter.

[14] In fact, any matrix that can be coerced to either of these types of matrices are legal for a triangular solve operation.

Chapter 5

MaTRiX+$^+$ Compiler

The MaTRiX+$^+$ compiler is a Compositional C++ (CC++) [3] package that constructs efficient parallel implementations for matrix computations. The compiler is written in an object-oriented style with the dual goals of extensibility and runtime efficiency. The concepts of objects and dynamic dispatch provide the flexibility for describing a compiler in terms of components that can be readily modified. Efficiency at runtime is obtained by using highly tuned numerical kernels as the target elements of matrix operation implementations.

This chapter describes the principle elements of the compiler and its runtime environment. We will begin with a brief discussion about CC++ and why we chose it as the underlying language. Then we will present the structure the MaTRiX+$^+$ compiler. Since the current system is a prototype (built on top of other prototypes), the focus is on the overall structure and the design decisions; the issues that are raised are those that should be invariant across implementations. Details about the underlying "substrate", the LAPACK++ package, will only be briefly discussed. For the runtime environment, the focus is the model of execution. This model is a data-driven version of single-program multiple-data (SPMD) that supports straightforward implementation, hiding of unnecessary execution detail, and efficient runtime characteristics. Again, specific code is avoided since it only serves to display idiosyncracies of the CC++ system.

5.1 Compositional C++

We had a number of requirements for choosing an appropriate implementation language. Foremost, was that the language support our threaded, data-driven SPMD model of computation that we will discuss later. Additionally, we wanted the language to look as much

like C++ as possible so that users would be able to use MaTRiX+$^+$ without special extra knowledge. Although there are a number of prototype extensions to C++ designed to deal with distributed computation [11][14], we found that CC++ was most appropriate[15]. It provides concepts that directly support our model of computation. Also, the CC++ language has a clean model that allowed us to encapsulate most of the language-dependent aspects in a small number of classes and function. Therefore, the majority of the MaTRiX+$^+$ compiler is in standard C++.

5.2 Compiler Elements

The MaTRiX++ compiler is composed of three major modules: matrix structures, matrix operations, and system support. Each module contains a number of interacting classes. As mentioned above, the organization is intended to support both extensibility and runtime efficiency. The following discussion will stress these features.

5.2.1 Matrix Structures

The matrix structure module contains a set of classes whose purpose is to support the declaration and instantiation of distributed, hierarchical and base matrix structures; Figure 5-1 on page 45 gives the organization of the principal classes[16]. The classes are divided into two groups: interface classes and auxiliary classes; the interface classes are the ones that are accessible by a user of the MaTRiX+$^+$ language; the auxiliary classes do all the behind the scenes work. The edges indicate the class that provides either the functionality or representation of some class element.

The interface classes (MTRX_Type, MTRX_Object, MTRX_Property, and MTRX_Proc_Mesh) capture the information expressed in a user's specification. The class MTRX_Type stores the information about the dimensionality, the property (in an internal format), and the distribution type. In addition, it allows sub-type operations through iterators. In comparison, the MTRX_Object class is only a wrapper for an internal representation class. Another simple class is MTRX_Prop; it serves only as a clean interface to the more complicated matrix property classes. Finally, the class MTRX_Proc_Mesh stores the necessary dimensionality information and provides access to the processor mesh descriptor.

The auxiliary classes[17] (MTRX__Object_Rep and MTRX__Prop) provide the internal representations for the interface classes[18]. The MTRX__Object_Rep class is to the MTRX_Type class; it provides both storage for information such as dimensionality, property,

[15] More information on CC++ can be found through http://www.compbio.caltech.edu.

[16] This figure doesn't show a number of minor classes such as the iterators and value accessors because their design is rather trivial.

[17] Classes with double underscores in their names are internal classes that are not accessible to the user.

[18] Both classes are actually considerably more complex than their abstract renditions shown in Figure 5-1; again, only the major elements are exposed.

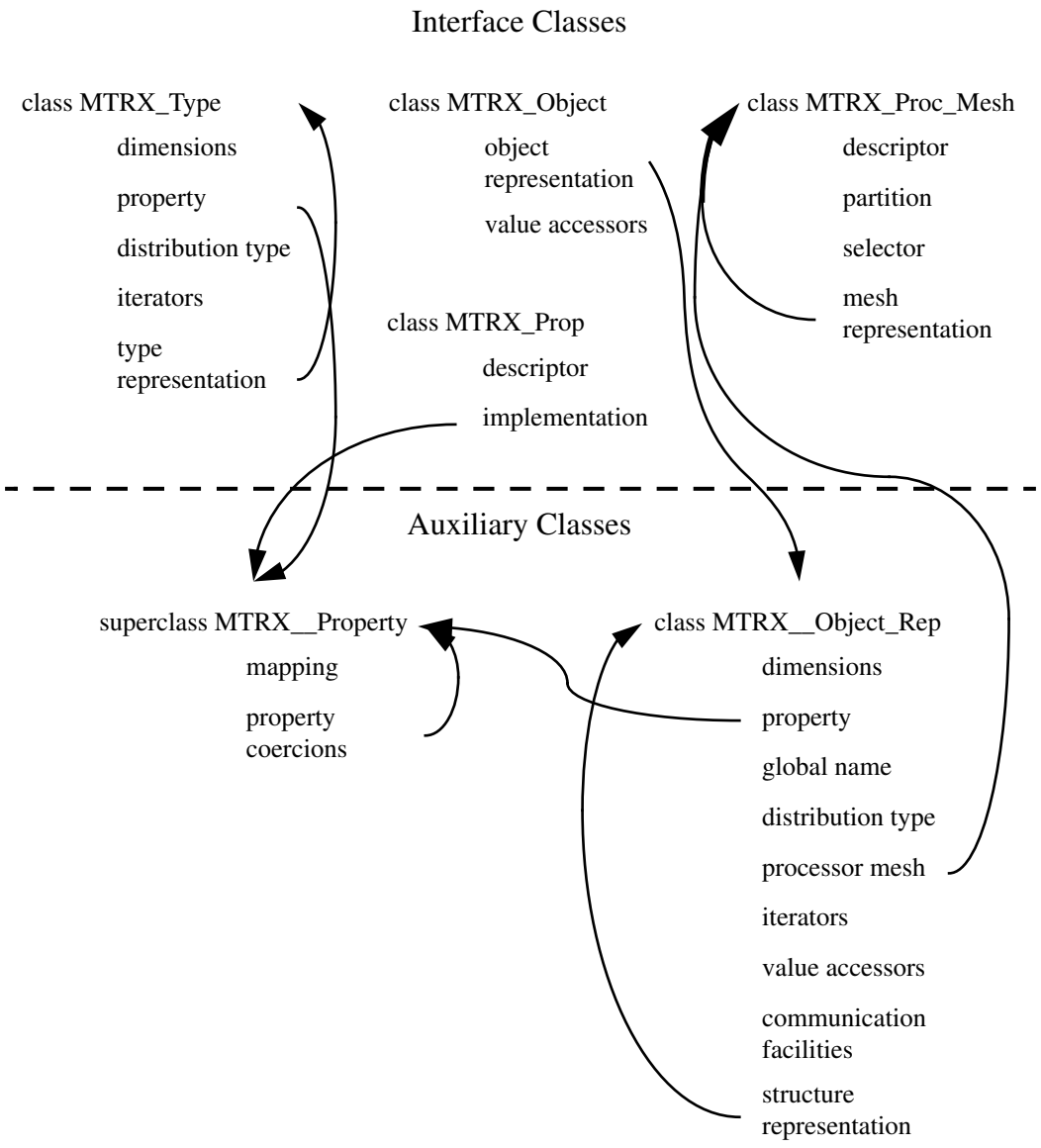


Figure 5-1. The organization of the principal classes for matrix structures. The classes with the prefix MTRX_ are user accessible whereas the class prefixed with MTRX__ are internal classes. The designation superclass indicates that there are a number of subclasses that are not shown.

and distribution and access via iterators. It differs in that it also has a global name, a process mesh descriptor, communication facilities, and value accessors. The most important feature is the global name that is associated with every matrix object element. It serves as a key to allow unambiguous communication of objects between processors. This communication is performed by the communication facilities which are in essence data packing and unpacking routines. Additionally, the MTRX__Object_Rep class has access to a collection of wrapper classes that provide the interface to the underlying representation and computational sub-

strate. Currently, this level is implemented by the LAPACK++ package. But any package[19] that provides matrix representations and type-specific implementations of matrix operations could be used with only minor changes to the MaTRiX+$^+$ compiler. The MTRX__Prop class is the superclass for all the classes for the specific matrix properties. These classes provide the functional capabilities of representational mapping and property coercions. The representational mapping function convert matrix indices to representation indices in order to support compressed storage. Property coercions supply a mechanism for performing property transformation at runtime.

The most striking feature of this design is the separation of the concept of matrix structure into a number of different classes: specifically, MTRX_Type, MTRX_Property, MTRX_Object, MTRX__Prop, and MTRX__Object_Rep. This division is motivated almost exclusively by reasons of modularity, information hiding, and extensibility rather than by the goal of efficiency. The divisions that we will discuss are the following: matrix structures into MTRX_Type and MTRX_Object, matrix objects into MTRX_Object and MTRX__Object_Rep, matrix properties into MTRX_Property and MTRX__Prop (and its subclasses), and types into MTRX_Type and MTRX_Property.

The separation of matrix structures into a type and an object is a purely functional decision. One can think of a matrix structure as having two phases to its existence: description and use; during the description phase, matrix types are used to capture matrix semantics; during the use phase, matrix objects provide an instantiation of types that allow representational and computational functions to be performed.

The partition of matrix objects into an object interface and an object representation is motivated by object-oriented programming style. It is simply good practice to make class interfaces as large as they need to be, and no larger. In the case of the MTRX_Object class, the user need only be aware of the capabilities of object instantiation and value accessing. The concepts of global name and communication facilities are hidden in the MTRX__Object_Rep class where they are not visible to the user. This separation has the additional benefit of shielding the user from any future changes in underlying representation.

A similar motivation suggests the separation of the property interface from the property representation. Primarily, it allows the seamless introduction of new matrix properties. Secondarily, it insulates the user from the exact names of the matrix properties. This latter benefit is important for maintaining the simple interface of the MaTRiX+$^+$ language.

Possibly the most subtle and important design decision was the separation of the type from property (and object from property). This decision has a number of implications. The obvious benefit is that of reuse of the property classes for both types and objects. But on the sur-

[19] In fact, the package doesn't even have to be written in C++. The LAPACK++ package is really a C++ interface to the Fortran Lapack and BLAS routines.

face, it appears that this benefit would be far outweighed by the possibility of implementing the type-subtype relationship among properties as a class hierarchy with inheritance. This approach seems the most natural and logical in the object-oriented methodology. If we made matrix operations and algorithms internal to such classes, we would apparently gain type coercion and type-specific algorithm dispatch for free.

Unfortunately, there exist a couple of minor drawbacks, and one major drawback. We will begin with the minor problems. First, in the case of type coercion, different related properties may have different representations and therefore pointer casting[20] will not be sufficient. Second, the interface of this matrix class would be gigantic; this would increase the possibility of errors and reduce the comprehensibility of the code. Third, the addition of new matrix operations and matrix algorithms would require the recoding of basic classes. There are a number of other lesser problems, but the major problem is that construction of type-specific implementations may not work in all cases.

The problem is that the C++ language definition does not allow ambiguous type conversions for overloaded functions. This problem is best understood with an example. Take the case of a unary matrix operation, OP, and the matrix properties classes diagonal, upper bidiagonal, and lower bidiagonal. Now we can view both upper bidiagonal and lower bidiagonal as superclasses of diagonal. Therefore there exist implicit type conversions from diagonal to both of its superclasses. Our problem arises in the case that there are algorithms for OP for upper bidiagonal and lower bidiagonal but not for diagonal. If OP is applied to a diagonal matrix object, since no algorithm exists for this property, another one must be selected via type conversion. The crux of the matter is that there exist two apparently equally good choices. There are no preference mechanisms in C++ and so this ambiguity is treated as an error. Due to this problem, the MaTRiX+$^{+}$ compiler employs its own unambiguous dispatch mechanisms that will be discussed next.

5.2.2 Matrix Operations

The matrix operations module, as the name suggests, provides the interface and the implementation of the well known matrix operations. Figure 5-2 gives the abstract structure of the major superclasses: MTRX_Operation and MTRX__Algorithm. The former provides the user interface and runtime dispatch mechanism for operations. The latter are the functional elements that are composed at runtime as type-specific implementations of the operations. Again, there is a division between the classes for the sake of both extensibility and the more important issue of implementability discussed above.

The MTRX_Operation class provides the functions for type management and implementation of matrix operations. For type management there are functions for compatibility

[20] In C++ it is possible to convert a pointer to an object of a class to a pointer to an object of a superclass. This information losing process is analogous to property coercion.

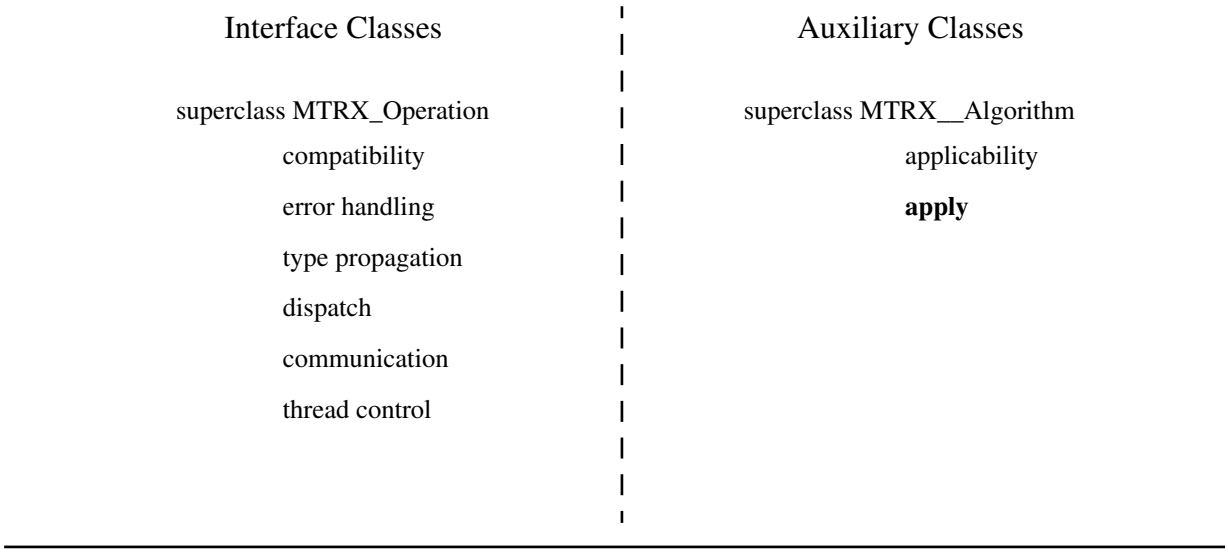


Figure 5-2. The primary superclasses for the matrix operation module.

checking and their associated error functions, and mechanisms for propagating type information to the results of the operation.

The dispatch, communication, and thread control functions are an integrated set of functions that support the runtime implementation of matrix operations in an SPMD model of execution. The efficiency of the dispatch function is crucial for the performance of the system. Although dynamic dispatch has a slightly greater overhead than normal function call, it can result in the choice of an algorithm that is orders of magnitude more efficient.

The current MaTRiX+$^+$ implementation trades space efficiency for time efficiency. It explicitly builds unambiguous dispatch tables. Runtime dispatch then requires only the constant cost of hashing a key in order to obtain an offset in a table. The communication functions determine how processors exchange information during the execution of an operation. Since the model of execution is data-driven, the process of communication serves to synchronize the set of processors. In comparison, the thread control functions provide monitoring information about the threads of execution within a processor. Also, thread control exposes possibilities for efficient execution through overlapping of computation and communication

The MTRX__Algorithm classes provide type-specific implementations of matrix operations. A very large number of algorithms are needed to implement all the possible instances of a matrix operation with typed inputs. Since not all the needed algorithms are represented in the MaTRiX+$^+$ compiler[21], the runtime system relies heavily on type coercion for handling the unsupported combinations of input types.

[21] For some operations the number of possible algorithms is quite enormous. Since there are about 30 different types in the current MaTRiX+$^+$ system, for the three input matrix multiplication, there exists 30^3 algorithms.

```
void LU_Tridiag_SDDom :: apply(LUFactorize& lufadd)
{
// Get number of diagonal elements
   int size = lufadd.num_diag_elts();
   MTRX_SubMatrix sm;
   for (int i=0;i<size;i++) {
// Perform LU on main diagonal block
       MTRX_LU(lufadd.A().ref(i,i),lufadd.F().ref(i,i);
// Do triangular solve on off-diagonal blocks
       MTRX_TriangSolve(lufadd.F().L().ref(i,i),
                        lufadd.A().ref(sm(i,i+1,i,size)),
                        lufadd.A().ref(sm(i,i+1,i,size)));
       MTRX_TriangSolve(lufadd.A().ref(sm(i+1,i,size,i)),
                        lufadd.F().U().ref(i,i),
                        lufadd.A().ref(sm(i+1,i,size,i)));
// Update next diagonal block
       MTRX_Mult(lufadd.F().L().ref(sm(i+1,i,size,i)),
                 lufadd.F().U().ref(sm(i,i+1,i,size)),
                 lufadd.A().ref(sm(i+1,i+1,size,size)),
                 lufadd.A().ref(sm(i+1,i+1,size,size)));
   }
// Perform LU on last diagonal block
   MTRX_LU(lufadd.A().ref(size,size),lufadd.F().ref(size);
}
```

Figure 5-3. The apply function for the algorithm class that implements the in place LU decomposition for tridiagonal, strictly diagonally dominant hierarchical matrix structures.

The structure of the MTRX__Algorithm class is specifically designed to facilitate dynamic dispatch with the possibility of type coercion. It provides functions that determine whether the algorithm is applicable given the types of the input objects to matrix operation. In addition there is a special member function called **apply**. This function contains the code that implements the algorithm. A representative example of the apply function for an algorithm for a hierarchical matrix structure is shown in Figure 5-3. Unsurprisingly, the code looks just like a MaTRiX+$^+$ program; there are selector functions applied to matrix objects and matrix operations; additionally there are reference objects which are matrix objects that refer to some rectangular sub-region of another matrix object. For algorithm classes for operations on base matrix structures the apply function is even simpler. It is merely a function call to the appropriate LAPACK++ routine.

5.2.3 System Support

The system support module provides all the control mechanisms for the model of execution and the dynamic aspects of the MaTRiX+$^+$ compiler, yet is almost transparent to the user. Figure 5-4 shows the principal abstract classes and some of their functionality. The classes can be divided into two groups: global (MTRX__System, MTRX__Network, and MTRX__Communication) and local (MTRX__Proc_Init and MTRX__Processor). The global classes, under the direction of a master processor, implement the administrative details of

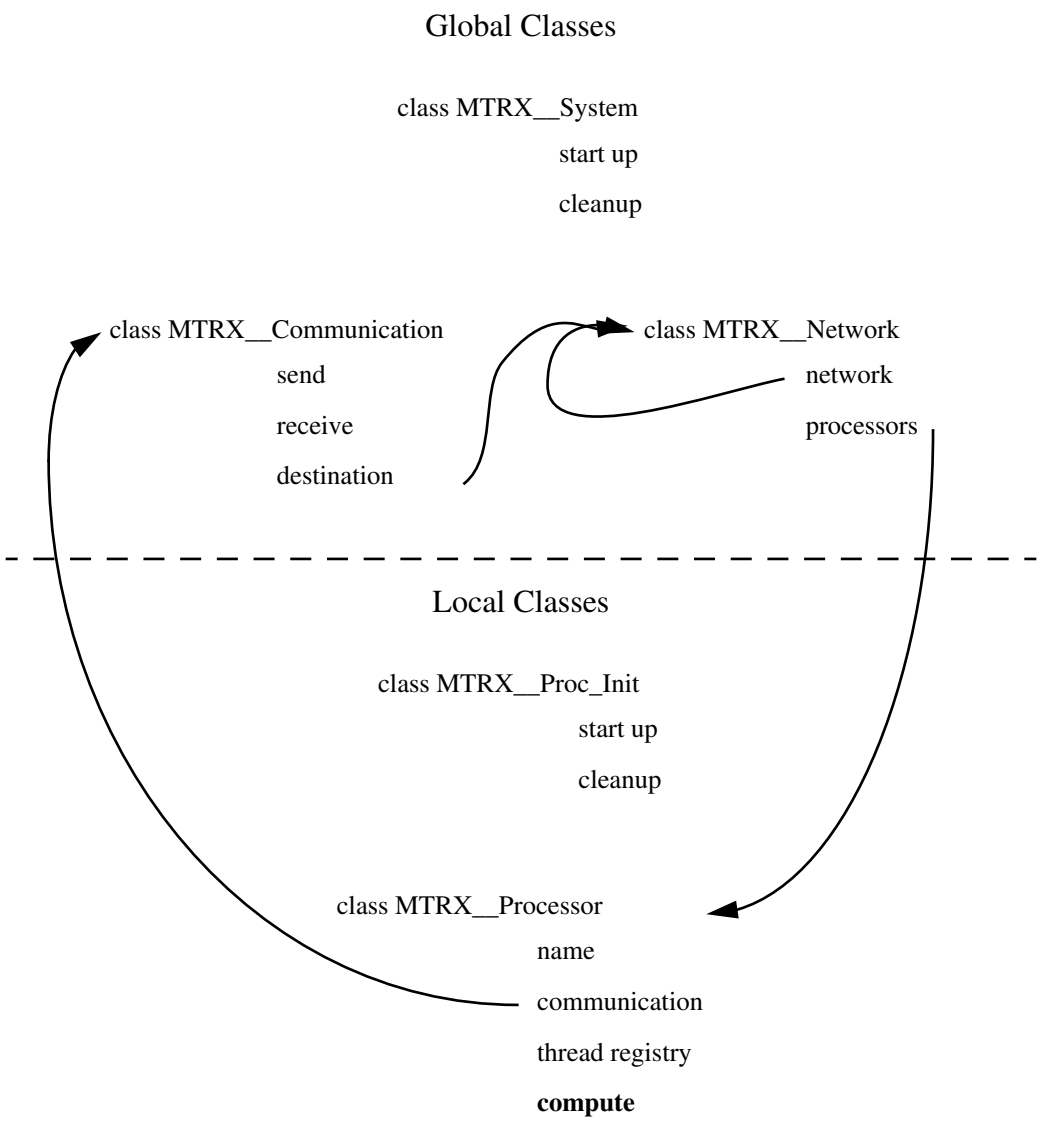


Figure 5-4. The abstract classes involved in system support.

initializing a mesh of SPMD processes. The local classes, that perform the matrix computations, operate on individual processors controlling the local threads of execution.

The global classes provide the bridge between the globally distributed computation and the local matrix computations. To a large degree this is all transparent to the user whose specifications include no explicit parallel flow-of-control constructs. The data-driven SPMD model of computation automatically exploits and controls parallelism.

It is the MTRX__System class that moderates the initial execution of a computation. Using the user's description the start up function creates a mesh of processors all executing

the specified executable on the chosen input files. After the completion of the computation, the cleanup function returns all the remaining memory used by the computation and terminates all the associated threads of execution.

The MTRX__Network class supplies the representation of the mesh of processing elements. Essentially, it is an all-to-all "graph" where the nodes are processes. The processes can be accessed by name by using the processor function.

The class, MTRX__Communication, provides the protocols for communicating on the edges of the "graph". Both synchronous and asynchronous sends and receives are supported. In addition there is a destination function for mapping a processor name to an actual process in the network.

The nodes of the "graph" are the local SPMD processes. They are defined by the class MTRX__Processor[22]. The most important member of this class is the **compute** function. This is the slot in which the user's specification is actually written. Another part of the class interface is the function associated with the thread registry mechanism. It provides some limited access to the state of the threads running on the local processor. Since the processors must be able to communicate with each other, there must exist functions that have access to global information. The name functions allow a processor to determine its own "name" and the "name" of other processors in the network. Additionally, there are communication functions that interface with the MTRX__Communication class objects.

Perhaps the most important local class is MTRX__Proc_Init. Its purpose is to manage the dynamic aspects of the MaTRiX+$^+$ compiler. Specifically, it controls the creation and destruction of

- the matrix property type system,
- the matrix operation interfaces,
- and the matrix algorithm registry[23].

All these objects must be created before the **compute** function of the MTRX__Processor class is run.

5.3 Runtime Environment

Since the MaTRiX+$^+$ compiler is a CC++ library package, the majority of its functionality is exploited at runtime. The goal is execution efficiency. In this section we examine how the runtime semantics of the MaTRiX+$^+$ system attempt to achieve efficient execution. We

---

[22] The term "processor" is perhaps a little misleading because a number of SPMD processes may be mapped to a single physical processor. Therefore, the "processors" should be thought of as virtual processors.

[23] These dynamic aspects of the compiler will be described in more detail in the following section.

will begin with an overview of the execution model of MaTRiX+$^+$ programs and then focus on the specific semantics of distributed matrix object construction and distributed matrix operation execution.

5.3.1 Data-Driven SPMD

The SPMD (single-program multiple-data) model of computation is a common target of parallel compilers [15][24]. Its definition is that every processor executes the same program but over different data. This model has the advantage of being both powerful and simple. A number of compilers use SPMD for implementing SIMD- style data-parallel computations on distributed memory architectures[24]. Their view is that a computation is composed of a linear sequence of sequential and data-parallel stages. The semantics of the sequential stages is that they are either executed identically on all processors or by some master processor. The data-parallel stages involve some pre-communication followed by the distributed computation and a barrier synchronization. This approach is simple, but it may lose opportunities for effective coarse-grained parallelism.

Our threaded, data-driven SPMD model of computation exploits parallelism at two conceptual levels. First, each processor executes the same program, but each may take different data-dependent paths through the program. Second, each processor may run a number of threads down its path in the computation. These threads within a processor may synchronize with each other and block on data that is needed from other processors. This judicious use of asynchrony is the key to exploiting both fine- and coarse-grained parallelism.

There are two properties of MaTRiX+$^+$ computations that make them amenable to this model of computation; first, matrix operation interfaces provide unambiguous information about which matrix objects are to be read and which ones are to be written; second, because both matrix structure and processor name information are accessible from the matrix object data structure, these send and receive events can be performed in an autonomous manner. We will discuss this issue further in Section 5.3.5.

5.3.2 Execution Overview

When executed, a MaTRiX+$^+$ program goes through a number of different stages. Figure 5-5 shows these major phases and the ordering relationships among them. The phases can be separated into two groups: administration and computation. The stages of system start up, processor initialization, processor clean up, and system clean up comprise the administrative components and were described in the previous section. The computation phases consist of type checking, object construction, and operation implementation. The following sections will discuss each of these in turn.

---

[24]This model of computation is also sometimes called SCMD (single-command multiple-data) [15].

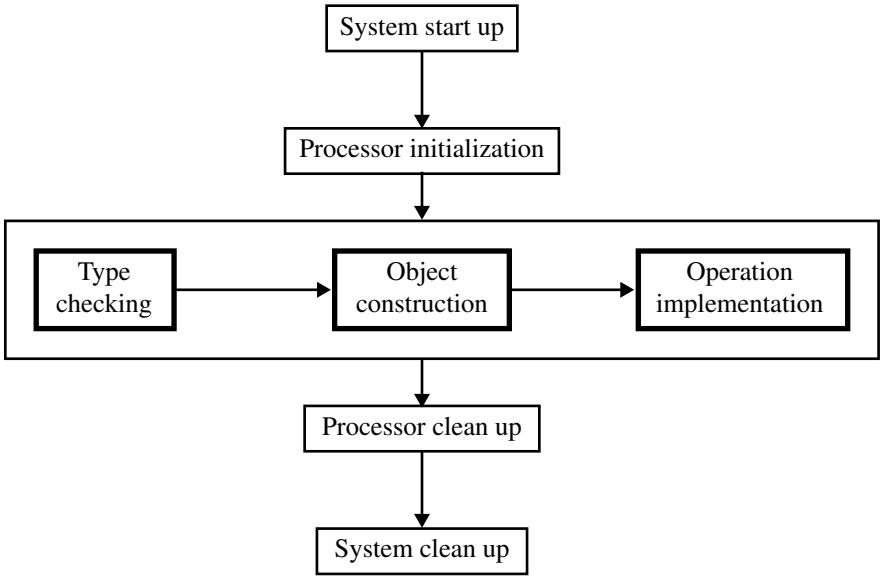


Figure 5-5. The execution phases of a MaTRiX+$^{+}$ program.

5.3.3 Type Checking

Type checking ensures that the matrix types in a specification are meaningful. It guarantees not only that the hierarchical structure is legal, but also that the matrix properties have been asserted in the correct manner. Checking the legal structure requires that the hierarchical structure represent the recursive partitioning of some matrix space (as defined in Chapter 3). A matrix property is legally asserted as long as its definition is not violated by the properties of any the elements of the structure. For example, if the property upper triangular is asserted about some hierarchical matrix structure, then if any of the elements in the lower triangle have a non-zero property, this will result in an error. Once a matrix type is type checked in these two ways, its type may no longer be changed. To do so results in a runtime error.

5.3.4 Matrix Object Construction

Matrix objects are instantiations of matrix types that provide runtime representations of hierarchical matrix structures. How the matrix objects are constructed can greatly affect the performance of the system. Also, the process must fit cleanly into the data-driven SPMD model of computation. With these goals in mind, we discuss the representation of matrix objects.

The data structure for matrix objects can be viewed as a tree with implicit edges as in Figure 5-6 on page 55. In this tree the nodes are matrix object descriptors and the edges are

pointers between descriptors; the leaves, since they represent base matrix objects, are objects that contain LAPACK++ data structures. Furthermore, based on the matrix type and the initial processor mesh, each node is annotated with a specific element of the processor mesh descriptor. The question is what semantics to assign to these annotations.

Clearly for a distributed-memory representation we are going to have to utilize the annotations to determine the matrix object representations that each processor constructs. As mentioned above, the goals for the distribution semantics are computational efficiency and ease of use in the data-driven SPMD model of computation. We will examine a bad distribution strategy and thereby arrive at a better strategy.

Consider the case that a processor represents some internal node, B, of the graph yet does not represent its parent node, A. First, allowing such distributions would greatly complicate the matrix structure representation. Second, any access to B will invariably require an access to A. So, the processor that represents B will have to perform a remote operation in order to access its own data. This overhead will result in unacceptable performance. Therefore, we can induce the rule that if a processor represents some node, it should also represent the node's parent. This suggests that the representation involves not only distribution, but also replication.

The natural distribution strategy that obeys our rule is that each processor constructs all matrix objects that are annotated with its processor number (name). Figure 5-6 illustrates the different elements of a matrix object that are constructed by a set of processors. One important result of the representation rule is that all processors (described in the initial processor mesh) construct all top level matrix objects defined in the user's program. This means that initially each processor knows the name of the processor that owns each matrix object element. Therefore every processor can determine where to send and from where to receive needed object information. This property is exploited in constructing efficient communication mechanisms. The matrix operation execution model must ensure that this property holds throughout the whole computation.

5.3.5 Matrix Operation Implementation

The task of matrix operation implementation is to produce type-specific, efficient, data-driven SPMD programs. In this model of execution, all processors execute the same code, yet different processors take different data-dependent paths through the program; this must all be achieved without the guidance of a master processor. Therefore, each processor must be able to determine independently the "role" it must take in the distributed computation. Since it may be necessary to communicate non-local data, the processors must also be able to determine to where they must send data and from where they will receive it. Furthermore, it is possible to perform a number of computational and communicational threads on the same processor. We will explore each of these issues in turn beginning with processor roles. The following is the notation that we will be using for the remainder of the discussion.

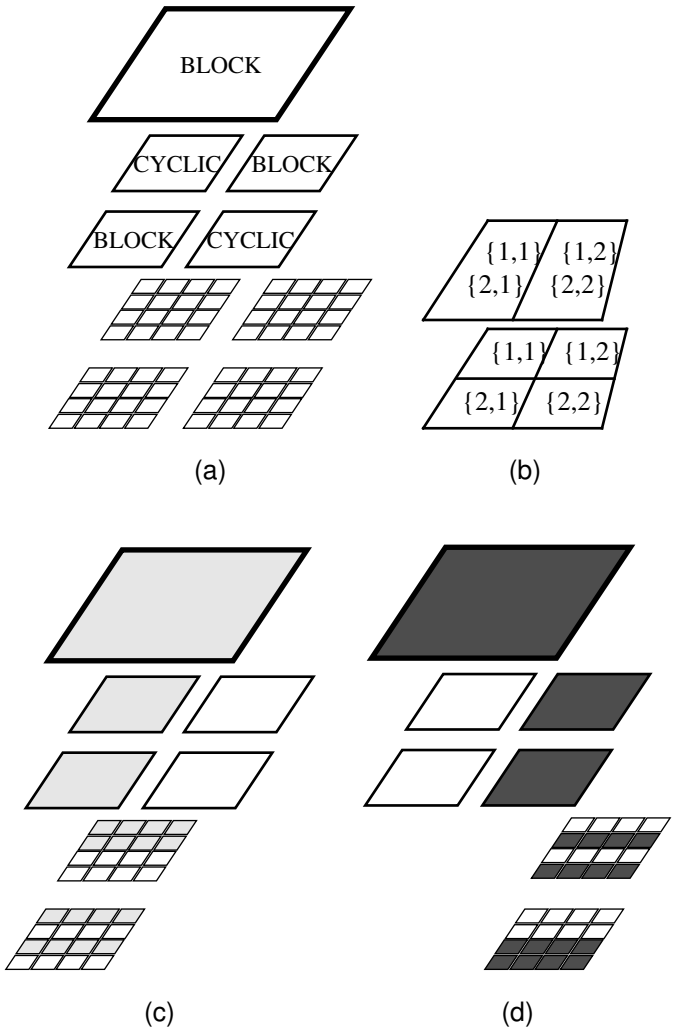


Figure 5-6. The local representations on processor {1,1} (c) and processor {2,2} (d) given a matrix (a) and a processor mesh (b).

Notation: The notation $M: (I_1, I_2, ..., I_n) \rightarrow R$, describes a matrix operation M that takes n input objects and produces one result object. In addition we define $\Psi_I = \Psi(I_1) \cup \Psi(I_2) \cup ... \cup \Psi(I_n)$, $\Psi_R = \Psi(R)$, and $\Psi_U = \Psi_I \cup \Psi_R$ where Ψ_I is the set of processors that own inputs to the operation, Ψ_R is the set of processors that own the result of the operation, and Ψ_U is the total set of processors involved in the matrix operation.

Processor roles

Processor roles are the different data-dependent approaches that a processor can take to implementing a matrix operation. We identify three roles named *reader*, *writer*, and *spectator*. For a matrix operation M, a processor $P_{i,j}$ is designated a reader if $P_{i,j} \in \Psi_I$ and $P_{i,j} \notin \Psi_R$, a writer if $P_{i,j} \in \Psi_R$, and a spectator if $P_{i,j} \notin \Psi_U$. The principal execution rule associated with the roles is that reader and writer processors perform the given operation and spec-

tator processors merely skip the operation. The justification for this rule is that if a processor owns no part of the matrix objects involved in a given operation, then the result of the operation can have no effect on the processor's local name space.

Role of communication

MaTRiX+$^+$ programs often require the communication matrix objects (or parts of matrix objects during) the execution of a matrix operation. In these programs three kinds of instances of matrix operations arise: matrix operations involving a reference matrix object, matrix operations involving only elements of matrix objects that point to hierarchical matrix objects, and matrix operations involving only elements of matrix objects that point to base matrix objects. Only the latter two kinds of operations may require communication. We will refer to these operations as *hierarchical element operations* and *base element operations*.

One way of performing communication for hierarchical element operations is for some subset of the processors in Ψ_I to send to the processors in Ψ_R a full description of all the input matrix objects. The disadvantage of this approach is that potentially very large messages are communicated between processors. In addition, large amounts of unnecessary matrix information may be sent. Because of these reasons, we opt for a hierarchical communication strategy[25].

Instead of communicating a whole matrix object, we only send the next level of description of the object. So for example in Figure 5-6 on page 55, the matrix object in (a) has three levels of hierarchy. Each element (a rectangular region in the figure) is a level of description in that it describes the type information about the elements which comprise it. Since we are only sending a partial description of the matrix object, we must now send descriptions of both input matrix objects and the result matrix object to all processors in Ψ_U. The reason that the result object must be communicated is so the processors in Ψ_I will be able to determine where to send remaining input object descriptions as the operation is hierarchically resolved.

For base element operations, it is only necessary to send input objects from Ψ_I to Ψ_R. This approach is sufficient because the operation has already been resolved down to the level of base matrix operations: no further communication can be necessary.

Models of communication

We now discuss a model for determining which processors must really communicate necessary matrix object information. This involves constructing a non-centralized communication strategy such that each processor can determine independently its send and receive events. As a result of using a regular hierarchically partitioned mesh of processors, this model

[25]This scheme is also more compatible with all the notions of hierarchical implementation.

Reader	Writer	Spectator

if (base element operation)
  send inputs to Ψ_R
else
  send inputs to Ψ_U
  receive inputs from Ψ_I
  receive result from Ψ_R
  perform matrix operation
	if (base element operation)
	  receive inputs to Ψ_I
	  perform matrix operation
	else
	  send result/inputs to Ψ_I
	  receive inputs from Ψ_I
	  perform matrix operation
		nop

Figure 5-7. Pseudo code for simple processor execution roles.

is straightforward. Given two matrix objects MO_1 and MO_2, there are only four different possible relationships between the processors of $\Psi(MO_1)$ and $\Psi(MO_2)$:

$$\Psi(MO_1) \equiv \Psi(MO_2) \quad (1)$$
$$\Psi(MO_1) \subset \Psi(MO_2) \quad (2)$$
$$\Psi(MO_1) \supset \Psi(MO_2) \quad (3)$$
$$\Psi(MO_1) \cap \Psi(MO_2) \equiv \varnothing \quad (4)$$

Here we are considering the communication of MO_1 to $\Psi(MO_2)$. In the cases of relationships (1) and (2), since $\Psi(MO_2)$ already has a copy of MO_1, no communication is necessary, For relationships (3) and (4), MO_1 must be communicated. Since all processors in $\Psi(MO_1)$ have a copy of the matrix object, any one of them could potentially send its description to the processors in $\Psi(MO_2)$. It is desirable for the sake of communication speed to use multicasts in lieu of point-to-point communication whenever possible.

Execution model

Given the description of processor roles and their associated communication schemes it is straightforward to induce the execution strategy shown in Figure 5-7. This execution model is simple and correctly performs distributed parallel matrix operations without any explicit global synchronization mechanisms. The problem with this model is that the implicit synchronization caused by communication events can cause unnecessary processor idle time. It is therefore desirable to determine a strategy that avoids the potential for loss of efficiency. The key idea of a more sophisticated strategy is to have several threads of execution on each processor. This allows the possibility of overlapping activities such as communication and computation. We briefly explore some of the sources of this thread parallelism and present a new strategy that takes advantage of some of it.

There exist two kinds of sources of parallelism in MaTRiX+$^+$ programs: between matrix operations and within matrix operations. The first level of parallelism exists between matrix operations in the user's program; the second level exists within the matrix operations themselves as they are implemented by the compiler. In principle, the first level of parallelism

Reader	Writer	Spectator

spawn thread
{
  if (base element operation)
  send inputs to Ψ_R
  else
  spawn thread
  {
  send inputs to Ψ_U
  }
  receive inputs from Ψ_I
  receive result from Ψ_R
  perform matrix operation
}
	if (base element operation)
	  receive inputs to Ψ_I
	  wait until all threads finish
	  perform matrix operation
	else
	  send result/inputs to Ψ_I
	  receive inputs from Ψ_I
	  wait until all threads finish
	  perform matrix operation
		nop

Figure 5-8. Pseudo code for threaded processor execution roles.

could be exploited through dataflow analysis of the user's program. But due to the nature of the typical scientific computations (large matrices and a small number of operations over them) this approach would only yield minimal useful parallelism. We will therefore focus on the second level of parallelism.

It is desirable to exploit parallelism within matrix operations in a manner that changes the model of execution as little as possible. Therefore, explicitly parallelizing the matrix algorithms is not an acceptable approach. Additionally, this methodology ignores processor role information that cannot be determined until runtime. The approach we will take is analogous to the database serialization constraint, that for each processor, all reader roles that precede a writer role in the sequential execution of the program must complete before that writer role is executed.

Figure 5-8 provides the pseudo code for this threaded "serialized" approach. In this approach, reader roles are always spawned on their own threads. This allows the possibility of parallelism between reader roles on the same processor; in many cases, this parallelism would not have been exploited in an explicitly parallel scheme. The disadvantage of the "serialized" method is that it cannot take advantage of possible parallelism among writer roles. A combination of the two methods would be a useful extension, but it would still not exploit the possibility of interleaved execution of interacting operations.

5.4 Summary

The MaTRiX+$^+$ compiler is a CC++ package that constructs efficient parallel, distributed memory implementations of matrix computations. The compiler is written in an object-oriented style with the dual goals of extensibility and runtime efficiency. Extensibility is gained from a modular design that allows the simple addition of new matrix properties, matrix operations, and matrix algorithms. Runtime performance in achieved through a data-driv-

en SPMD model of execution that uses highly tuned numerical kernels as the target elements of matrix operation implementations.

A runtime execution consists of a collection of autonomous node programs that operate asynchronously ordered by send and receive events. Internally the node programs are threaded “serialized” execution streams that overlap activities of computation and communication. The actual computation is performed by type-specific matrix algorithms that are recursively composed via an efficient dispatch mechanism.

Chapter 6

Application Examples

Matrix operations arise in almost all scientific applications. The premise of this research is that in many of these computations the matrices are both very large and highly structured. The purpose of the MaTRiX+$^+$ system is to provide a user with a powerful set of abstractions for describing this structure in such a form that the MaTRiX+$^+$ compiler can exploit it in the generation of efficient parallel implementations of hierarchical matrix computations.

In this chapter we will briefly examine three example problems: electrical power systems, heat transfer, and submarine acoustics. The goal is to understand the matrix structure that arises in each problem area and then to encode it in the form of a MaTRiX+$^+$ program. In addition, we will explore alternative problem formulations and examine the effects on the resulting MaTRiX+$^+$ specifications and their computational properties. For the sake of clarity we will make simplifying assumptions that will allow us to focus on the task of mapping physical problem structure to declarative matrix structure.

6.1 Matrix Structure and Ordering

In scientific applications, matrix structure is the result of the complex interaction among the modeling method, the physical problem, and the ordering strategy. The modeling method describes an abstraction for associating variables (unknowns) with features of the problem. When applied to a physical problem, these unknowns are assigned physical locality. Ordering is the process of mapping these "physical" unknowns to their places in a matrix.

Although the mathematics and the physical problem determine the number, the relationships among, and the values of the unknowns, ordering is the step that decides what ma-

trix structure will arise. Poor orderings will result in a scattering of the values of a sparse matrix in ways that inhibit structure exploitation and effective parallelism. Good orderings expose hierarchical structure that is both exploitable and exposes effective parallelism. In this chapter, starting in section 6.3 on page 61, we will explore a number of variations on a single ordering theme. For each ordering variation, we will examine the issues of how it simplifies the problem, if it produces useful structure, and whether it exposes effective parallelism.

6.2 Electrical Power Systems

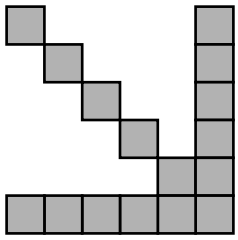
Electrical power system models [19] are used to simulate the response of power generation and distribution networks to component failures. These systems can be viewed as graphs where the nodes are electrical buses and the edges are electrical transmission lines. In turn, these graphs can be represented by matrices with nodes as diagonal elements and edges as off-diagonal elements. Due to low density of high-voltage lines, and the high density of local networks, the matrix is irregularly sparse.

Parallel matrix solving methods are necessary because these matrices can be large (10,000 to 50,000 unknowns), and it is desirable to be able to solve them in real-time. The key to these methods are ordering techniques that transform the sparse matrices that arise in power system applications into the “arrow” (block-diagonal-bordered) form shown in Figure 6-1 (a). In addition to having the arrow shape, the matrix is also symmetric and positive definite due to the modeling method. This allows the possibility of using a Cholesky factorization ($A=L*L^T$) which requires half the computation of the standard factorization. Another computational advantage of the arrow structure of this matrix type is that each block on the diagonal (except the last) can be solved independently. To reiterate, the structure of this matrix is a result of the interaction among the modeling method, the physical problem, and an ordering method.

The MaTRiX+$^+$ code for describing this application is shown in Figure 6-1. The key observation that should be made, is that the code is simple yet flexible. It is simple because the two levels of structuring of the matrix can be described in less than fifteen lines of code. But it is also flexible because it is parameterized by the block sizes. Therefore, it is a generic routine for solving a large number of power system problems. In the next example we will examine how different ordering strategies can be explored easily in MaTRiX+$^+$ programs.

6.3 Heat Transfer

Heat transfer is an example of many applications that model propagation through a medium. The mathematics results in partial differential equations that are solved via discretization over the physical space being modeled. One commonly used technique for solving the discretized equations is a finite difference method in which the physical problem is represented as a mesh of nodes and the differential operators describe relationships between nodes. An example of a differential operator, a 5 point stencil, is shown in Figure 6-2 (a). In a matrix



(a)

```
void MTRX_Execute :: execute()
{
// Input block sizes for diagonal elements
   int num_of_blocks;
   infile >> num_of_blocks;
   int sizes[num_of_blocks], loop, last, lastsize;
   for (loop=0;loop<num_of_blocks;loop++) {
        infile >> sizes[loop]; }
   last = sizes[num_of_blocks - 1];
   lastsize = sizes[last];
// Describe the matrix property
   MTRX_Property PSprop(ARROW,SYMM,POSDEF,DIAG);
   MTRX_Type PStype(num_of_blocks,num_of_blocks, PSprop);
// Describe the main diagonal
   for (loop=0;loop<num_of_blocks;loop++) {
        PStype.set_type(loop,loop,
            *(new MTRX_Type(sizes[loop],sizes[loop],MTRX_DOUBLE));}
// Describe the bottom and right side of the arrow. Since the matrix
// is symmetric, we only have to describe one of them.
   for (loop=0;loop<num_of_blocks-1;loop++) {
        PStype.set_type(last,loop,
            *(new MTRX_Type(lastsize,sizes[loop],MTRX_DOUBLE));}
// Instantiate matrix object and matrix factor
   MTRX_Object PSobject(PStype);
   MTRX_Factor PSfactor(PSobject);
// Perform the factorization
   MTRX_Cholesky(PSobject,PSfactor);
}
```

(b)

Figure 6-1. A power system application. The arrow matrix, (a), for the model, and the associated MaTRiX+$^+$ code, (b).

form, the nodes map to diagonal elements and the relationships map to off-diagonal elements. Due to locality principle of nodes only interacting with neighboring nodes, the matrix is very sparse. We will use a square mesh of nodes (Figure 6-2 (b)) to explore some possible ways to describe this sparsity.

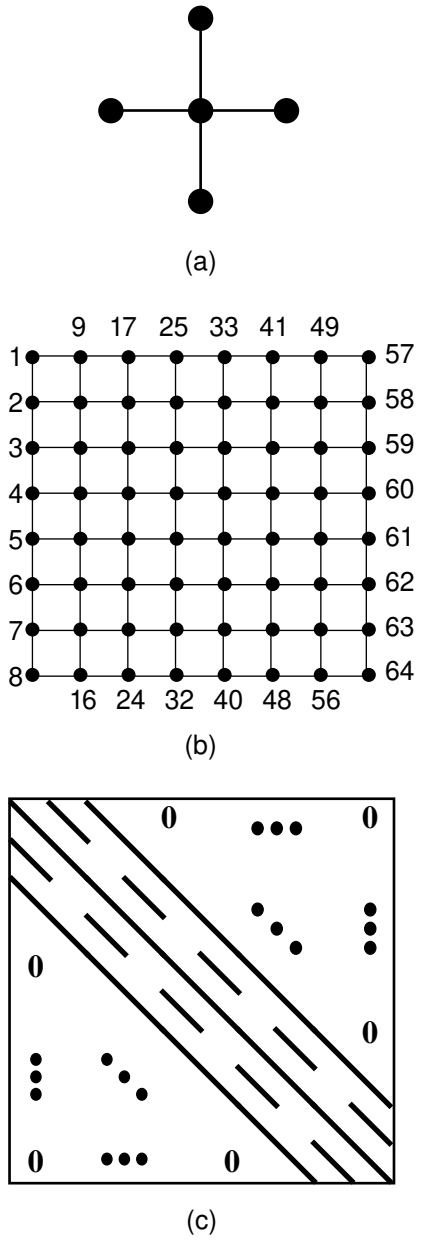


Figure 6-2. A finite difference example. A 5 point stencil is shown in (a), and a regular mesh of nodes is shown in (b) with a top-to-bottom, left-to-right ordering scheme, and the resulting matrix in (c).

6.3.1 Sequential Slicing Approach to Ordering

An ordering method for this mesh is an assignment of indices to nodes of the mesh. One natural ordering is the top-to-bottom, left-to-right assignment shown in Figure 6-2 (b). The matrix resulting from the ordering has a banded structure (Figure 6-2 (c)) that cannot be straightforwardly described with a typical matrix type.

We can make this structure much clearer by using the same ordering, but by viewing it hierarchically. In this approach, we "slice" the mesh into columns that are numbered left to right. Internally each column is numbered top to bottom. This numbering, shown in Figure 6-3 (a), is hierarchical in that the column numbers refers to regions, and therefore blocks in the matrix, and the internal number to nodes, and therefore scalar elements in the matrix. In the resulting hierarchical matrix the columns become the elements on the diagonal of the matrix, and the interactions between columns become off-diagonal elements. Since columns only interact with their neighboring columns, the actual structure of the top level of the hierarchical matrix is tridiagonal.

To determine the structure at the next level of the matrix, we must describe the interactions within a column and between columns. Within columns, nodes only interact with their consecutive neighbors; this results, yet again, in tridiagonal structure, type **A**. Between columns, only adjacent pairs of nodes interact; this results in diagonal structure, type **B**. Figure 6-3 shows both the slicing and the hierarchical structure of this approach. The MaTRiX+$^+$ shown in Figure 6-4 exemplifies the simplicity of this structure.

Execution analysis

Unfortunately, this formulation will not have very good run-time performance. First, during a factorization operation (LU or Cholesky), this matrix structure will suffer from large amounts of fill-in. That is, values that were previously zero become non-zero. Second, there is almost no exploitable parallelism. We will now explore an alternative ordering strategy that helps alleviate both of these problems.

6.3.2 Recursive Slicing Approach

The recursive slicing approach is a problem decomposition method. The principal step is to divide the current problem into three parts, a central physical slice, an actual column of nodes, and two "abstract" slices on either side. These abstract slices are the sub-problems that will be resolved recursively in the same manner. The ordering strategy is also recursive. We will number the region on the left first, the region on the right second, and the physical slice last. The result of the first slice is the 3-by-3 arrow matrix shown in Figure 6-5. Such a structure is a consequence of the fact that we have divided the problem into two non-interacting sub-problems. We will first determine the structure of the elements of the matrix and then examine its effect on the manner of writing the MaTRiX+$^+$ program.

Recursive slicing

Consider the hierarchical structure of each of the seven non-zero elements of the matrix in turn (Figure 6-6). Since we are recursively partitioning, elements 1,1 (type **A**) and 2,2 (type **A**) are going to have the same arrow structure as the matrix itself. Element 3,3 (type **F**) represents the interactions of the nodes within a physical slice and therefore has the tridiagonal structure that we have seen before in Figure 6-3 (b). The structure of elements 1,3 (type

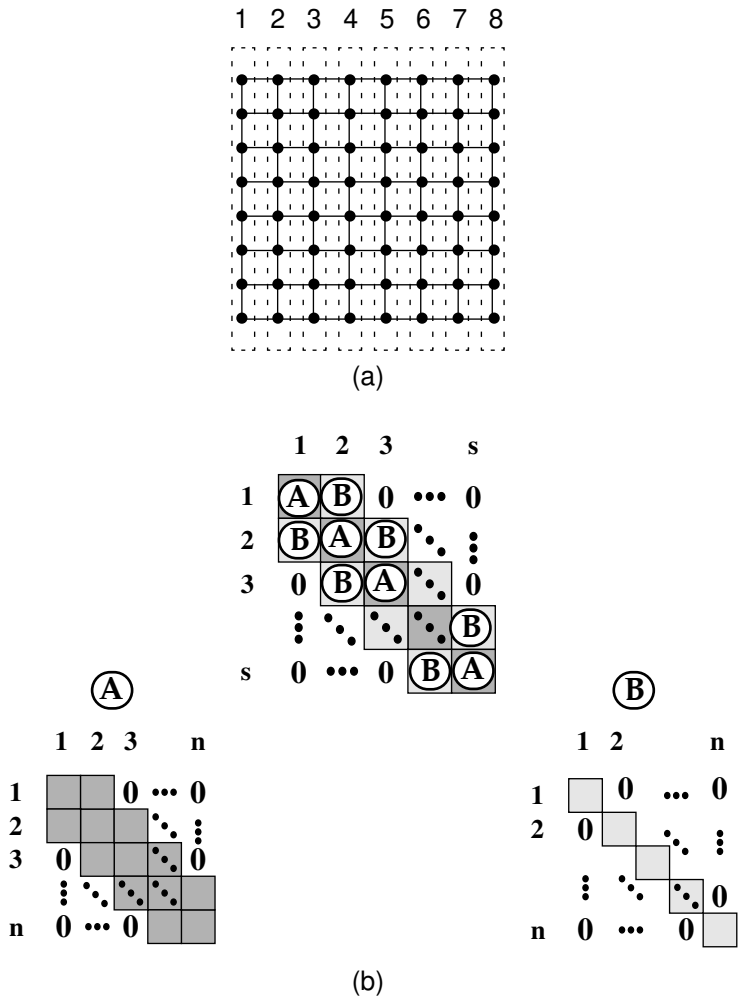


Figure 6-3. The sequential slicing approach: (a) shows the numbered slices and (b) describes the tridiagonal hierarchical matrix structure composed of sub-types **A** and **B**.

D), 2,3 (type **E**), 3,1 (type **B**), and 3,2 (type **C**) all describe the interaction between a physical slice and an abstract one. Since the structure is symmetric, we will only concern ourselves with the structure of elements 3,1 (type **B**) and 3,2 (type **C**). We will explore the structure of 3,1 (type **B**) by considering the result of a recursive slice of region 1.

We know from the physical problem and the modeling method that only the "rightmost" column of the abstract slice can interact with the physical slice, but we have yet to determine its numbering. Notice, that slicing always partitions a region into three sub-regions in which the rightmost column is always in the sub-region numbered second. Therefore, the structure of element 3,1 (type **B**) is a 1-by-3 matrix where elements 1,1 and 1,3 are zero matrices and element 1,2 is recursively this same structure, **B** (which bottoms out in base structure **B**). The structure of element 3,2 (type **C**) can be determined in the same manner through the "leftmost" relationship.

```
void MTRX_Execute :: execute()
{
// Input the number of slices and the block size
   int num_of_slices,size;
   infile >> num_of_slices;
   infile >> size;
// Describe the matrix properties for the tridiagonal, symmetric,
// positive definite elements and the diagonal elements.
   MTRX_Property TriSPDprop(SYMM,POSDEF,TRIDIAG);
   MTRX_Property Diagprop(DIAG);
// Define the finite difference type (the top level).
   MTRX_Type FDtype(num_of_slices, num_of_slices, TriSPDprop);
// Create a diagonal selector.
   MTRX_Diag diag;
// Describe main diagonal.
   FDtype.set_type(diag(0),
        *(new MTRX_Type(size,size,TriSPDprop,MTRX_DOUBLE));}
// Describe the lower and upper diagonals. Since the matrix
// is symmetric, we only have to describe one of them.
   FDtype.set_type(diag(-1),
        *(new MTRX_Type(size,sizes,Diagprop,MTRX_DOUBLE));
// Instantiate matrix object and matrix factor
   MTRX_Object FDobject(FDtype);
   MTRX_Factor FDfactor(FDobject);
// Perform the factorization
   MTRX_Cholesky(FDobject,FDfactor);
}
```

Figure 6-4. MaTRiX+$^+$ program for sequential slicing approach.

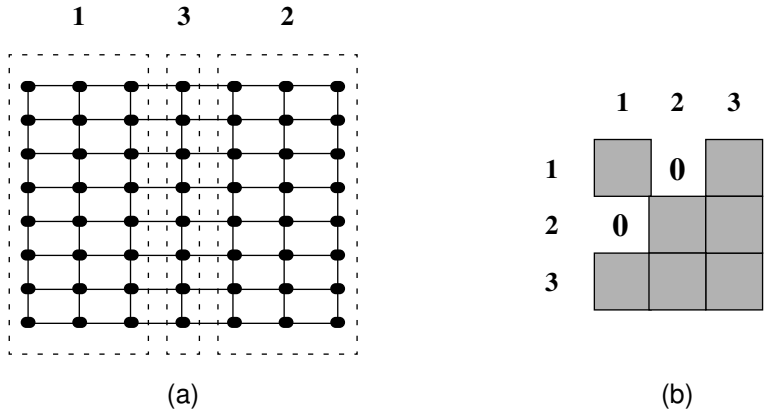


Figure 6-5. An abstract slice (a), and the resulting arrow matrix (b).

Recursive specifications

The MaTRiX+$^+$ code associated with this application must capture the recursive structure of the matrix types. The approach is to define type generators rather than just type declarators (as has been done up to this point); type generators are meta-type declarators that determine element types based on some user-determined condition, such as whether the struc-

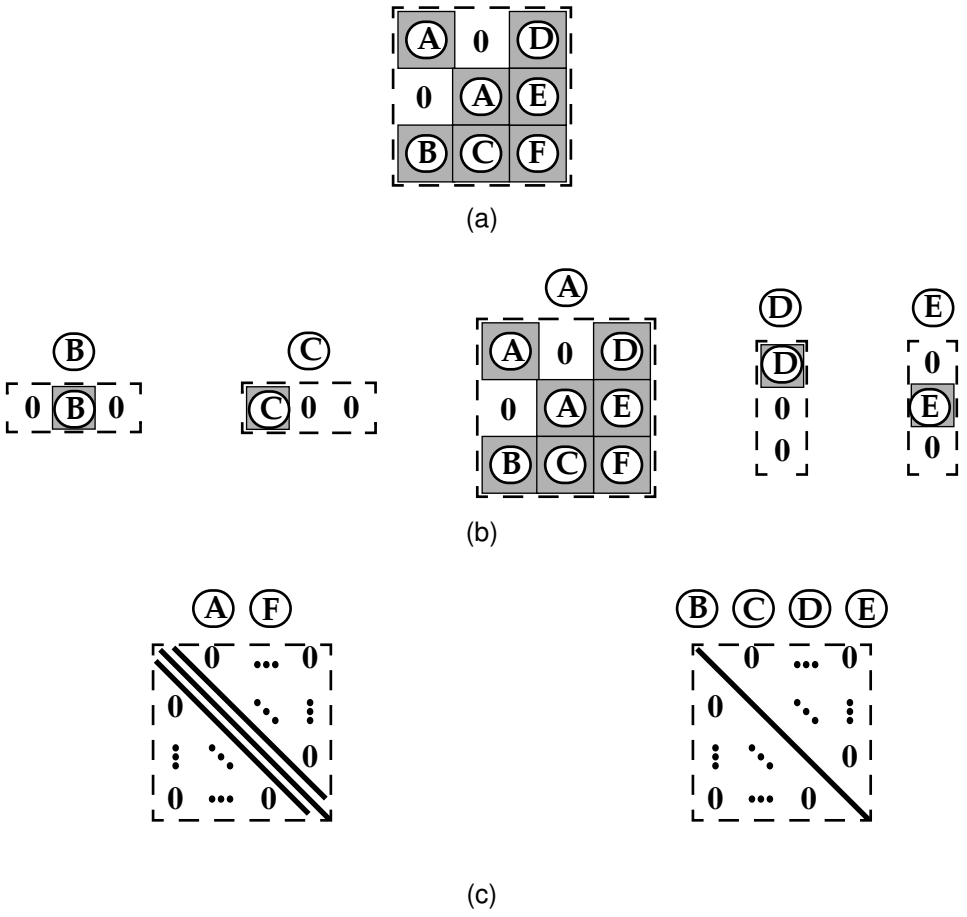


Figure 6-6. Hierarchical matrix structure of recursive slicing approach: (a) the top level arrow matrix, (b) the recursive types, and (c) the base types.

ture is hierarchical or base. Consider the recursive arrow structure that is declared by the MaTRiX+$^{+}$ code fragment in Figure 6-7. This routine returns either an arrow structure or a base matrix based on whether the slice is abstract or physical.

Execution analysis

Although this recursive formulation exposes considerable parallelism that was not available in the sequential slicing approach, it still is unable to exploit much of the parallelism that does exist. The problem is that the physical columns are the same size as the physical problem and therefore can be enormous. In other words, we have only exposed parallelism in one dimension of the physical problem. Clearly, we must determine a way to slice these regions also. The next section examines 2-D slicing as a solution to this problem.

```
// Type generator for recursive arrow structure
MTRX_Type*
ArrowTypeGenerator(int numSlices, int baseRSize,
   int baseCSize, int& newRSize, int& newCSize) {
// base case
   if (numSlices == 0) {
        newRSize = baseRSize; newCSize = baseCSize;
        return BaseDiagGenerator(baseRSize,baseCSize); }
   else {
        MTRX_Type *arrowtemp;
// Describe the matrix property for the arrow.
        MTRX_Property arrowProp(ARROW,DIAG,SDDOM);
        MTRX_Diag diag; MTRX_Element el;
        int& newr,newc;
// Describe main diagonal.
        arrowtemp->set_type(diag(0),
             *ArrowTypeGenerator(numSlices-1,baseRSize,baseCSize,
                                 newr,newc));
// Describe the lower edge
        arrowtemp->set_type(el(2,0),
             *EdgeGenerator(numSlices-1,ELT20,baseRSize,baseCSize));
        arrowtemp->set_type(el(2,1),
             *EdgeGenerator(numSlices-1,ELT21,baseRSize,baseCSize));
// Describe the right edge
        arrowtemp->set_type(el(0,2),
             *EdgeGenerator(numSlices-1,ELT02,baseRSize,baseCSize));
        arrowtemp->set_type(el(1,2),
             *EdgeGenerator(numSlices-1,ELT12,baseRSize,baseCSize));
// Describe the arrow tip
        arrowtemp->set_type(el(2,2),*TipGenerator(baseRSize));
   }
        newRSize = newCSize = 2 * newr + baseRSize;
        return arrowtemp;
}
```

Figure 6-7. MaTRiX+$^+$ code fragment for recursive slicing structure.

6.3.3 2-D Recursive Slicing

A 2-D slicing scheme partitions a physical space in two dimensions not only one. There are a number of possible ways in which to do such a partition. One obvious approach is to use a 1-D recursive slicing and then afterwards to recursively slice the columns. Unfortunately, this scheme does not produce any appreciable benefits in terms of parallelism. The reason is that this slicing only effects the structure in the “tip” (element 3,3) or the arrow. During a factorization, this region suffers fill-in and therefore much of the zero structure is lost. Therefore, we will develop an approach that attempts to place more of the structure in a form that is exploitable.

2-D recursive slices

The key to the one-dimensional slicing approach was that each physical slice (vertically) divided the problem into two non-interacting sub-problems. The idea of the 2-D method is to physically slice (horizontally) each of these subregions into two new non-interacting sub-problems. Therefore, after these two steps, the size of the sub-problems is one-quarter that of the original problem: halved in size in each dimension. A recursive version of this approach involves alternating the steps of vertical and horizontal slices until some terminal physical slice is reached; Figure 6-8 (a) shows an example of three steps in such a two-dimensional slicing. Not only does this two-dimensional slicing scheme extract more opportunities for parallelism, because each slice exposes two independent sub-problems, but it also reduces the size of the undesirable arrow "tip" region by three-quarters each step. We now examine how to describe this structure which is shown in Figure 6-8 (b-d).

We know that vertical slices result in our familiar 3 by 3 arrow matrix (type **V**). If our ordering strategy for horizontal slices is top region first, bottom region second, and physical slice third, then these also will result in 3 by 3 arrow matrices (type **H**). In addition, it is clear that physical slices (type **S**), whether vertical or horizontal, result in tridiagonal base matrices. This leaves only the interaction regions, elements 3,1 (types **A** and **D**), 3,2 (types **C** and **E**), 1,3 (types **A´** and **D´**), and 2,3 (types **C´** and **E´**), to consider. Since the structure is symmetric, we only have to determine the structure of elements 3,1 and 3,2. As in the description of the 1-D recursive slicing, we will focus on the structure of element 3,1.

We actually have to consider two kinds of element 3,1's: those arising in vertical slices, type **A**, and the those arising in horizontal slices, type **D**. We will begin with the vertical case since we have already examined it previously for the 1-D slicing. As before, only the "rightmost" nodes of region 1_1 can interact with the physical slice. Now consider the effect of a horizontal slice (#2) of region 1_1. This divides the type into three regions all of which contain some portion of the "rightmost" nodes.

We note two interesting observations about this specific interaction. First, even though there are three regions, there are only two different kinds of interactions: an abstract region with a part of a physical region (sub-regions 1_2 and 2_2) and a horizontal physical slice with a vertical physical slice (sub-region 3_2). Second, each of these sub-regions interact with a disjoint piece of the physical slice: sub-region 1_2 with the "top half", sub-region 2_2 with the "bottom half", and sub-region 3_2 with the "middle node." This latter interaction results in a matrix with a single non-zero entry (at whichever index is associated with the last node of the horizontal slice and the middle node of the vertical slice). We will focus on the more exploitable structure of the interaction between abstract slice 1_2 and physical slice 3_1. Fortunately, this can be done straightforwardly; the introduction of another slice (#3, vertically) results in (as we observed in 1-D recursive slicing) region 2_3 being the only region that contains a portion of the "rightmost" nodes. This pattern repeats as we cycle through the alternating vertical and

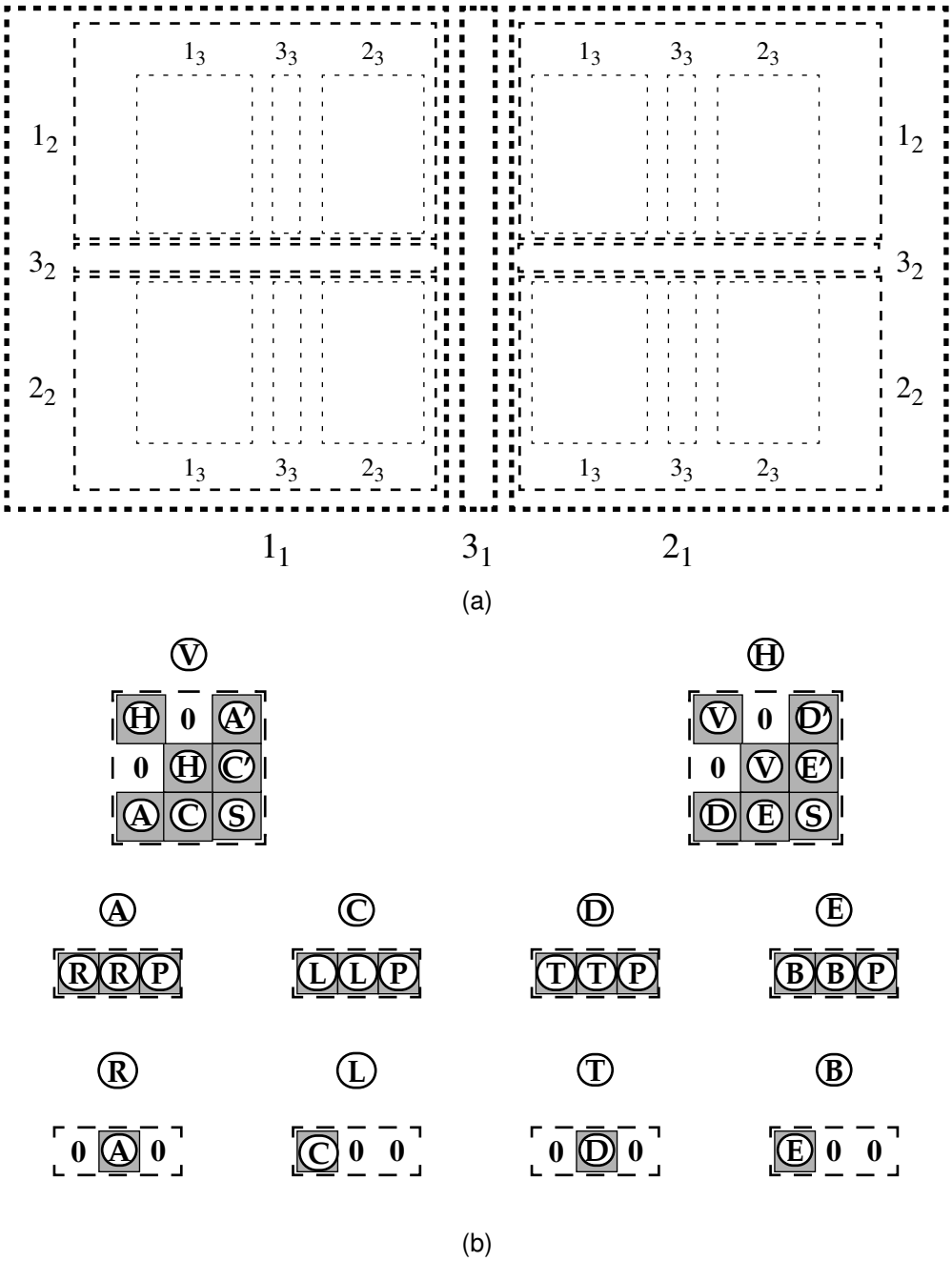


Figure 6-8. 2-D recursive slicing: (a) the physical slicing and (b) the resulting recursive types.

horizontal slices. Using this same analysis we can determine the structure of types **C**, **D**, and **E** by substituting the words “leftmost”, “bottommost”, and “topmost”, for the word “rightmost.”

6.3.4 Summary

In finite difference methods, the interaction among the discretized mathematical model, the physical problem, and the ordering method results in complexly structured sparse matrices. Using the MaTRiX+$^+$ language it is possible to construct hierarchical matrix types that capture the structure of these matrices. Moreover, it facilitates the exploration of various ordering strategies with the goal of obtaining better representational and computational characteristics. In the next section we will explore whether the matrix structuring techniques we have developed carry over to the more complex modeling mathematics of finite elements.

6.4 Submarine Acoustics

In Chapter 1 we presented the application problem of modeling the acoustical properties of a submarine. We will now address the problem of describing the matrix structure resulting from modeling the shell of the submarine using a finite element method.

6.4.1 Finite Element Methods

The finite element method [21][22] is an alternative to the finite difference method. Instead of representing a physical space by a mesh of nodes, a mesh of “elements” are used. Figure 6-9 shows an example of an element; this element consists of four “corners”, four “edges”, and a “bubble.” Not only is the composition of an element more complex than a node, but also its connectivities are much more complicated (Figure 6-9 (b-d)). In addition, there are a greater number of unknowns associated with the different components of an elements. For a given polynomial order p (the degree of the polynomial used to model the element) and the number of equations n (2 for two-dimensional modeling), each corner has n unknowns, each edge has n*(p-1) unknowns, and the bubble has n*(p-1)^2 unknowns. Therefore, for a two-dimensional physical space with p=6, the matrix resulting from an element is 98-by-98. We will now examine the effect of the added complexity of the finite element method on our ordering strategies.

6.4.2 Slicing Finite Elements

With the finite difference method we were able to determine slicing strategies that separated the modeling of the physical problem into independent sub-problems. The 2-D recursive slicing scheme was especially attractive in that it exposed substantial useful parallelism. The question is whether this same slicing approach works with finite elements. We will examine a mesh of elements to demonstrate that in fact it does.

Consider the mesh of elements in Figure 6-10 (a). Each row and column consists of either an edge-node-edge sequence or a bubble-edge-bubble sequence; the reason for this choice will become apparent very soon. First we must determine what constitutes a physical slice. The natural choice is an edge-node-edge sequence because it is the simplest means for dividing the mesh into two non-interacting sub-regions. When a slicing is performed, there

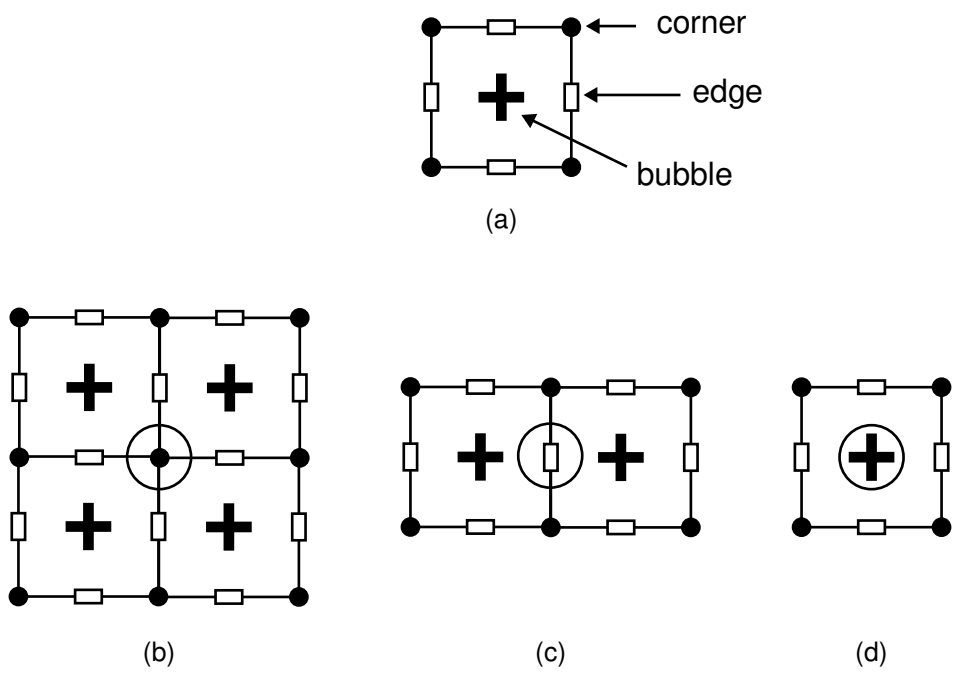


Figure 6-9. A finite element. (a) shows an element. The connectivities are shown in (b) (corner), (c) (edge), and (d) (bubble).

will be elements that remain that are not parts of physical slices; we will call these *primitive regions*.

A vertical slice (#1 in Figure 6-10 (a)) results in the all too familiar 3-by-3 arrow matrix. Due to the symmetric nature of the finite elements, a horizontal slice (#2) will also yield the 3-by-3 arrow matrix. In fact, the structure of the matrix will only differ from the finite difference matrix (2-D recursive slicing) in the structures that concern physical slices and regions: the physical slices, the interaction between physical slices, the primitive regions, and the interaction between primitive regions and physical slices. Since our physical slices are always edge-node-edge sequences, there will only be one kind of physical slice matrix type. If we number consistently, there will be only one matrix type associated with primitive regions (a suggested numbering is given in Figure 6-10 (b)). Also, there will only be only one (parameterized) type of physical slice interaction type, and four kinds of primitive region - physical slice interaction type.

Now that we have established that 2-D recursive slicing is an effective ordering technique we can attempt to use it for capturing the matrix structure of the submarine model.

6.4.3 Slicing the Submarine

It is not immediately obvious how to apply the 2-D recursive slicing technique to the submarine. Due to the physical structure of the submarine, a cigar-like shape, it is easy to

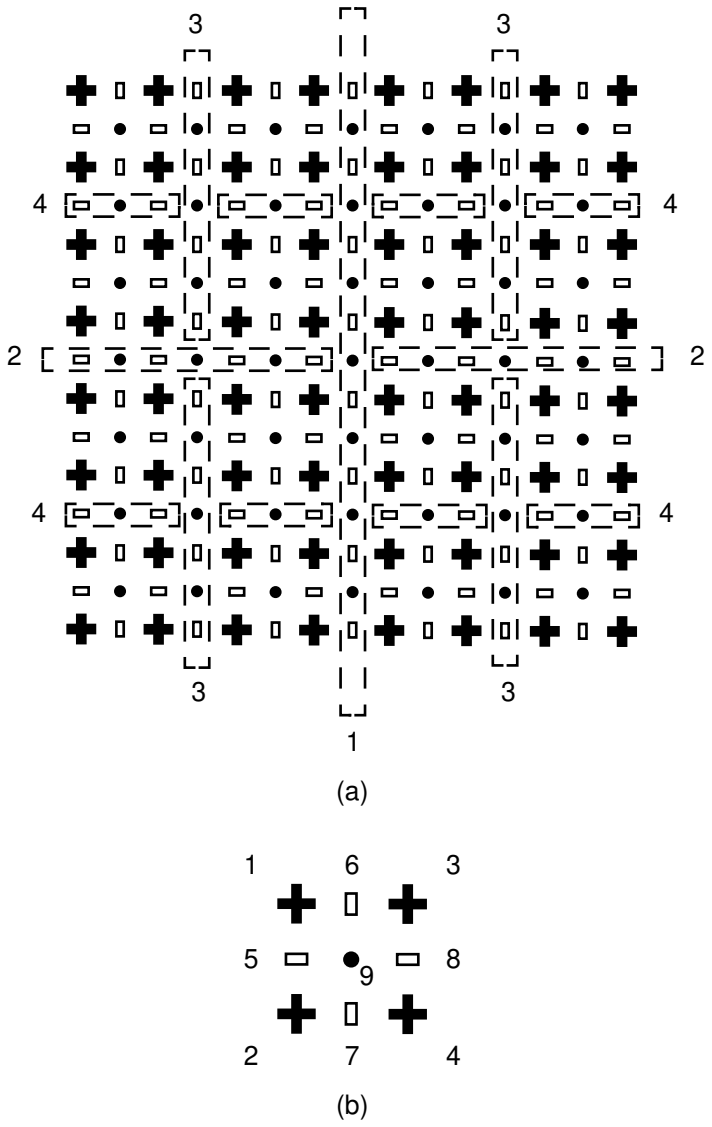


Figure 6-10. A mesh of elements: (a) a 2-d recursive slicing and (b) the numbering for the base physical region.

imagine that one could encounter strange problems with both physical slices and primitive regions. For example, one problem that could arise is that physical slices interact in strange ways at the ends of the submarine. Another problem is that the ends of the submarine could cause strangely shaped primitive regions to arise. Clearly it is desirable to somehow simplify the problem so that we can more straightforwardly apply a slicing strategy.

Up to this point we have been using slicing as an ordering mechanism. But at a higher level of abstraction, it is a problem reduction method; it breaks big problems into smaller subproblems. Therefore, we could think of performing some “abstract” slicing on the submarine to simplify the problem before applying slicing in the typical manner. Figure 6-11 shows such a simplifying slicing.

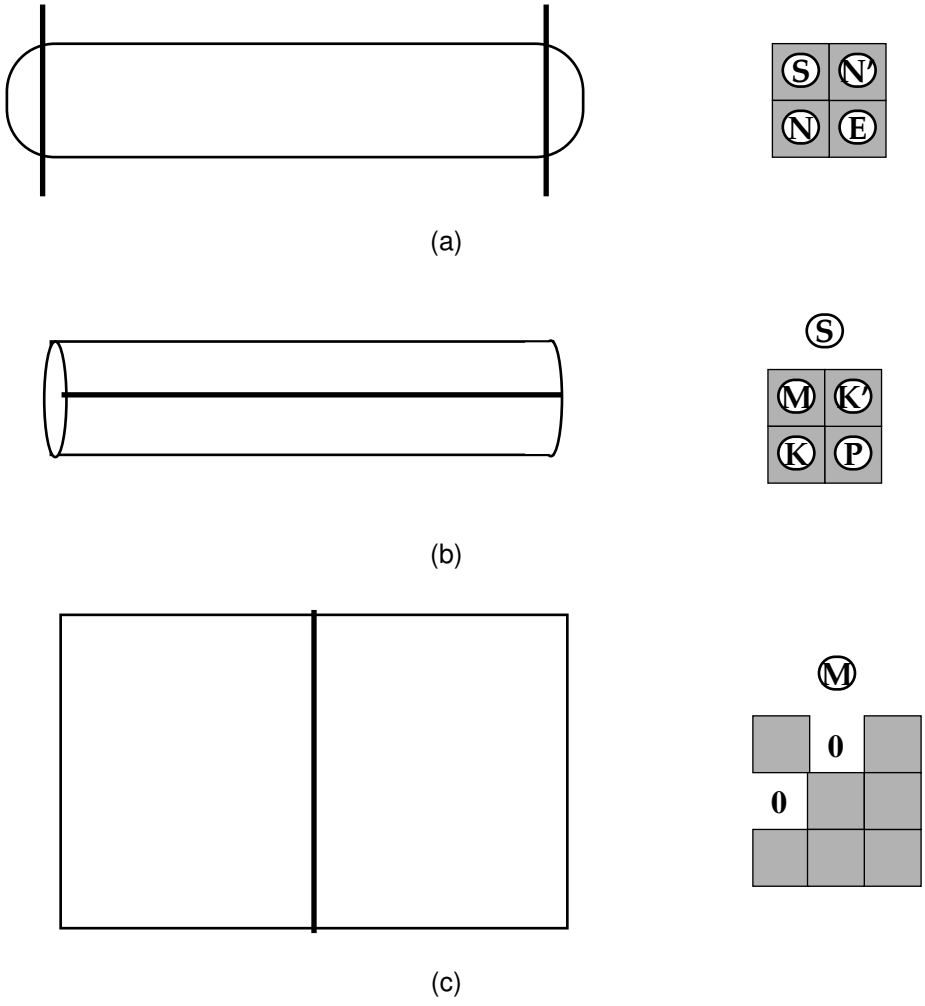


Figure 6-11. Abstract slicing of the submarine: the slicing of the ends and its matrix type (a), the slice of the resulting cylinder and its matrix type (b), and the final flat mesh of elements with an initial vertical slice.

The first slices remove the troublesome ends of the submarine (Figure 6-11 (a)). This results in a 2 by 2 matrix with the remaining portion of the submarine in element 1,1 (type **S**). The shape of this remaining portion is now a cylinder. If we now introduce a single longitudinal slice (b), then we are left with our familiar mesh of elements (c) which we can order using 2-D recursive slicing. Not only does this approach simplify the majority of the submarine description by reducing it to our familiar mesh of elements, but it also reduces the task of describing the structure of the ends, the longitudinal slice, and the interactions among the main parts, into clearly separated sub-problems.

6.5 Summary

The complex interaction among the modeling method, the physical problem, and the ordering of unknowns can result in extremely complicated matrix structures. The MaTRiX+$^+$ language provides the user with a simple syntax for describing this structure through hierarchical matrix types. Unfortunately, for a number of reasons, straightforward type descriptions may not always be appropriate computationally. The typed structure may be destroyed or unexploitable by the computation algorithm. Also, the type formulation may not provide opportunities for parallelism. We have shown how the MaTRiX+$^+$ system supports the development of the mapping between ordering strategies over the physical structure, and the hierarchical type of the resulting matrix. These strategies elegantly expose exploitable structures. Furthermore, in the case of the submarine application, we were able to identify an ordering strategy that converted the original complex problem into three, much simpler, subproblems. In the next chapter we will quantify the advantages of exploiting hierarchical type information.

Chapter 7

Experiments

The purpose of the MaTRiX+$^+$ system is to demonstrate the utility of hierarchical matrix typing. The principal metric for evaluating the success of the system is run-time performance. Since the current implementation is a prototype that is built on top of other prototype systems, optimized performance is an unrealistic goal. Instead, we intend to show that simple hierarchical types specifications in the MaTRiX+$^+$ language yield substantial performance benefits. In addition, we investigate the ability of the system to exploit available parallelism.

This chapter discusses the experimental methodology for evaluating the MaTRiX+$^+$ prototype. The first section describes the platform on which the experiments were run. The second section presents the sequential experiments, followed by the parallel experiments in the third section. The final section discusses the conclusions drawn from the experiments.

7.1 Experimental Platform

The experiments were run on a network of IBM RS6000 3BT workstations running AIX 3.2.5. A total of 17 such machines are connected by a FIDI local area network. The run-time environment consists of CC++ running on top of Nexus and DCE threads.

7.2 Sequential Experiments

The purpose of the sequential experiments is to examine the advantages of hierarchical type information and to demonstrate that a variety of different type specifications can be captured and exploited in the MaTRiX+$^+$ system. The first experiment investigates the advantage of having a large number of specific types in the system. It answers the question of whether many of the types that specify locations of zeros, such as tridiagonal, upper triangu-

lar, and arrow diagonal, are extraneous since we have a zero type. The second experiment explores the ability of the MaTRiX+$^+$ system to effectively exploit useful hierarchical type information. For this experiment we use a finite difference method with a 5-point stencil to compare the performance of the type-specific MaTRiX+$^+$ codes with LAPACK and BLAS routines. The third experiment explores another variation on this theme by examining the use of a 9-point stencil.

7.2.1 Zero Experiment

It could be claimed that the variety of matrix properties for hierarchical matrix structures that specify locations of zeros are unnecessary since they can be subsumed by the zero property. We demonstrate in this experiment that these properties are indeed useful. The experiment is very straightforward; we executed two versions of the electrical power systems application (section 6.2 on page 80). One version specifies the type of the hierarchical matrix as arrow-diagonal and the other version specifies unstructured and constructs all the appropriate zero blocks. For a 51-by-51 block matrix with block sizes of 20-by-20, the first version executed in a time of .15 seconds while the other "type-less" version took 27 seconds. This substantial difference in execution time is due purely to the large number of unnecessary function calls being perfomed over zero matrices.

7.2.2 5-Point Stencil Experiments

In this experiment we explore an approach to the solution of the system of equations specified by a finite difference method using a 5-point stencil. The primary goal is to demonstrate that hierarchical type information, if used appropriately, can lead to substantial increases in performance. The matrix we will solve is a sparse banded system (see Figure 6-2 on page 83 and Figure 6-3 on page 85).

The most obvious approach is to use a direct method involving an LU decomposition. But as we observed in 6.3.1, fill-in would almost nullify the benefit of the type information. Therefore, one would expect the execution time of the MaTRiX+$^+$ code to be almost equivalent to that of the LAPACK band LU routine. This is indeed confirmed through experimentation.

Another approach to solving the system of equations is to use an iterative method in which the principal operation is a matrix-vector multiply. This approach has two advantages: the type information can be fully exploited and there is no fill-in during the operation. We examine the execution times of the matrix-vector multipy over a range of matrix sizes generated from a number of different mesh sizes (see again Figure 6-3 on page 85). The physical mesh consisted of either 63 (problem #1) or 126 (problem #2) columns of nodes with the number of rows of nodes ranging between 50 and 200; the resulting matrix sizes ranged from 3150-by-3150 to 25200-by-25200. We compare the execution times to those of the BLAS (level 2) band matrix-vector multiply[26].

The execution times for these runs are shown in Figure 7-1. The MaTRiX+$^+$ implementations achieve considerably faster execution times for all but the smallest problem sizes. The largest speedup is in excess of a factor of 5. Also, two important facts can be deduced from the slopes of the graphs. First, the benefit of type-specific information increases with the size of the problem. And second, since the overhead introduced by the MaTRiX+$^+$ system is constant within the runs of problem #1 and #2, the low slope indicates that the majority of the execution time of the code is spent in non-computational routines. This suggests that an optimized version of the MaTRiX+$^+$ system would achieve considerably greater performance on this problem.

7.2.3 9-Point Stencil Experiments

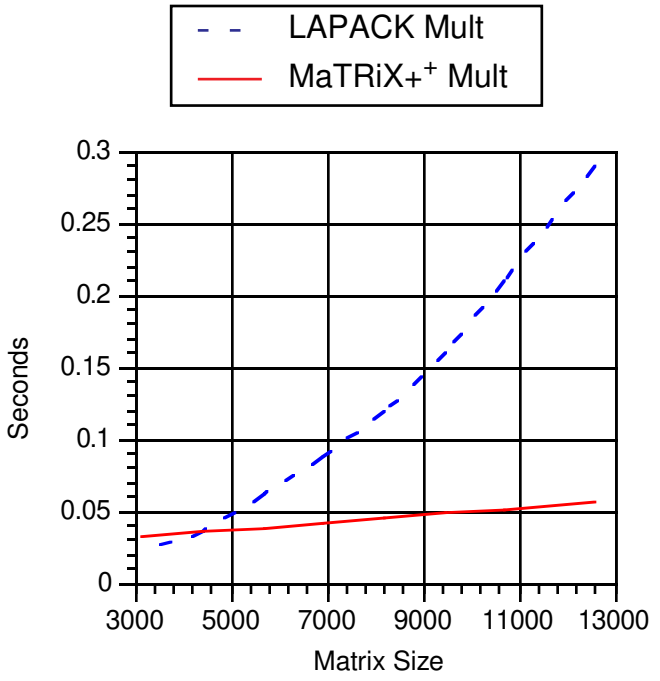
In the next set of experiments we move from a 5-point stencil to a 9-point stencil (Figure 7-2 on page 80). The purpose is to demonstrate that specification changes can be easily handled in MaTRiX+$^+$. In fact, the modification of the MaTRiX+$^+$ type specification is completely trivial: the elements on the lower and upper diagonals of the tridiagonal structure become tridiagonal structures themselves. On the other hand, it is not clear whether a hand-coded routine for the previous problem can be applied to this new problem without considerable recoding.

7.3 We used the same set of matrix sizes as were used in the 5-point stencil runs. The results of the 9-point stencil experiments are shown in Figure 7-3 on page 81. The pattern of the results is similar to the 5-point results. The main difference is that more of the execution time is spent in computational routines. This has the effect of moving the cross over point with the BLAS routine to a slightly larger problem size. Once again, an optimized version of the MaTRiX+$^+$ system would greatly improve this performance.**Parallel Experiments**

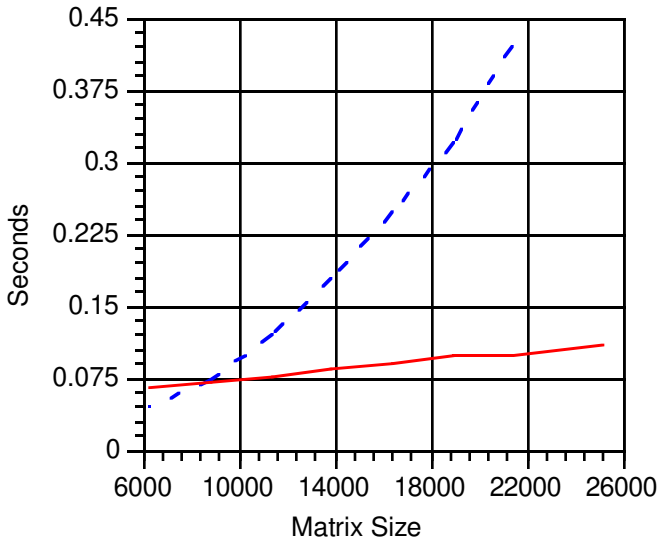
The sequential experiments demonstrated the utility of hierarchical type information. The goal of the parallel experiments is to show that parallelism can be automatically extracted from the executions of MaTRiX+$^+$ specifications. To obtain very effective parallelism a number of obstacles must be overcome. First, the platform must be able to parallel computations in a reasonable manner. In other words, the ratio computational speed on one processor to communicational speed among processors must small enough that it is indeed possible to achieve parallel speed-ups on problems of various sizes. Also, it is necessary that one be able to utilize the processors free of competition for resources with other users; contention for processor, memory, disk, and network bandwidth can have a disasterous effect on performance.

---

[26] One could clearly hand-code a more efficient routine for this special matrix, but it is specifically such an approach that this research attempts to provide an alternative to. Writing a complicated, type-specific routine is time consuming and error-prone task that usually results in code that is neither portable (in terms of efficiency) across architectures nor reusable. Therefore, it is not unreasonable to expect application engineers to resort to well-known, well-tested library routines.



(a)



(b)

Figure 7-1. Execution times for the 5-point stencil problems #1 (a) and #2 (b).

In addition to the platform and concerns, there are execution issues of good data distribution, load-balancing of computation, and efficient communicational strategies. Without successfully dealing with everyone of these platform and execution obstacles, it is simply not possible to obtain efficient parallel execution times.

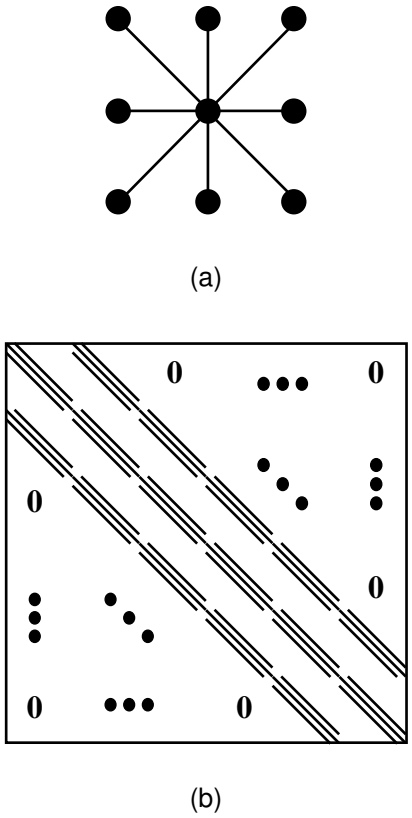


Figure 7-2. A 9-point stencil (a) and the resulting tridiagonal, tridiagonal matrix structure.

We faced a number of these problems. The network of RS6000's that we use has a high ratio of computation to communication speed for all but massive applications. Also, we continually compete with other users performing parallel executions of their own. In addition, since the MaTRiX+$^+$ system is constructed on top of CC++, we have no control over low-level details of communication and thread control. Therefore, it is no surprise that the parallel experiments point out a number a number of weaknesses in the MaTRiX+$^+$ system and the underlying CC++ and operating system mechanisms for communication. We ran the following set of experiments on the power systems application in order to expose some of these issues. We specifically choose this application because it contains a tremendous amount of exploitable parallelism.

7.3.1 The Naive Model

For the first experiment we simply executed a version of the specification in Figure 6-1 on page 82, on one processor and then on two processors. The problem size was 10260-by-10260 in the form of a 513-by-513 block arrow matrix where the blocks were of size 20-by-20. On the one processor run we obtained a real time of 116.4 seconds where only 5.8 seconds were spent in computational routines. This large disparity is due to both the paging that occurred as a consequence of the size of the matrix, and the fact that other processes obtained

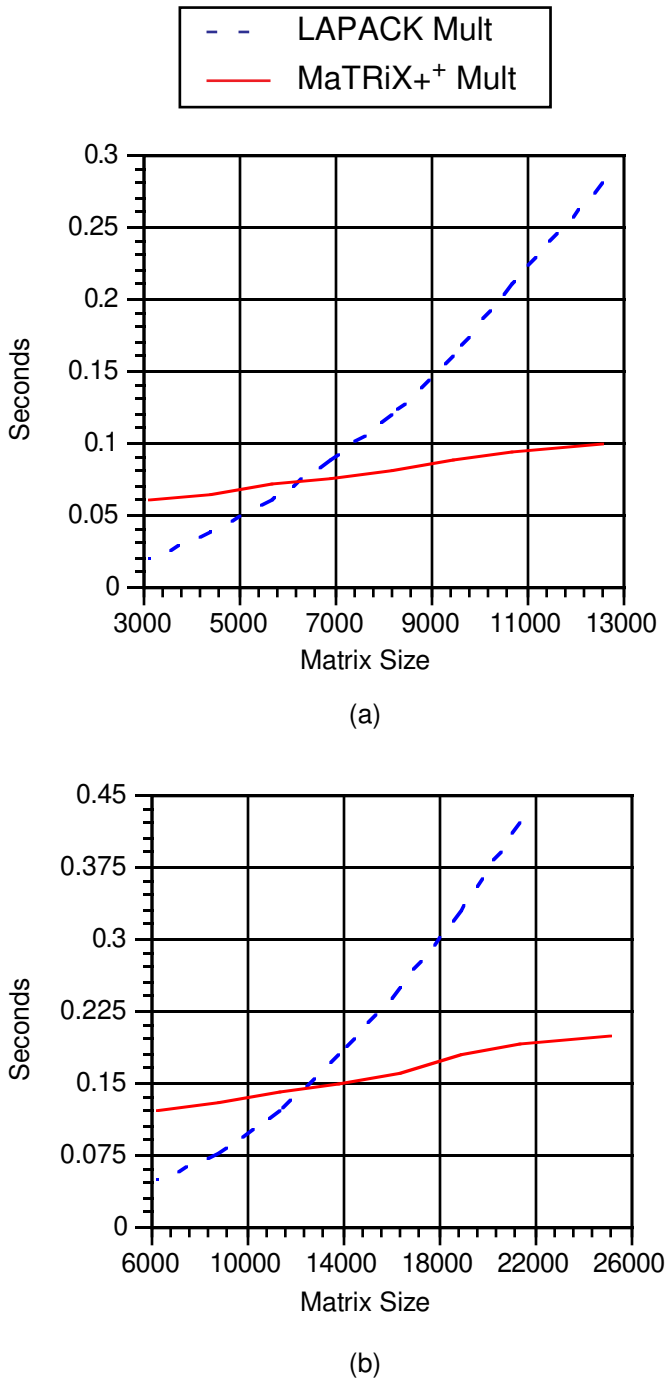


Figure 7-3. Execution times for the 9-point stencil problems #1 (a) and #2 (b).

the processor. On the two processor time was 132.4 seconds where 105.2 seconds were spent in computational routines. This difference in computations time cannot be explained purely by paging. We observe that processor 1 owns the “arrow tip”, and therefore has to perform a sequence of 512 matrix multiplications. The most obvious explanation is that the huge num-

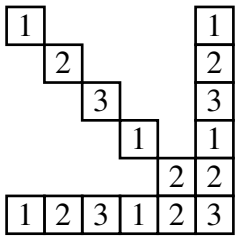


Figure 7-4. A SHELL distribution of an arrow matrix over three processors.

ber of frequent messages sent to processor 1 greatly interferes with its ability to perform its computations. We next explored an approach to reducing the frequency of the message traffic.

7.3.2 Block Approach

The block approach adds another level of structuring over the arrow-diagonal matrix. The result is another arrow-diagonal matrix with fewer, larger blocks. The idea is that a larger amount of computation can occurs before each communication event. Unfortunately, it is impossible to achieve useful data distribution with the HPF-style distributions supported directly in the MaTRiX+$^{+}$ system. Therefore, we invented a data distribution, called a SHELL distribution, that is more appropriate for arrow types. A shell consists of an element on the diagonal and an associated element on the right and bottom side of the arrow tip. In a SHELL distribution, shown in , shells are distributed in a round-robin fashion over processors.

We performed the following sequence of runs using the SHELL distribution blocks:

[1] 2 processors with 2, 4, 8, 16, 32, and 64 blocks (Figure 7-5 (a)),

[2] 4 processors with 4, 8, 16, 32, and 64 blocks (Figure 7-5 (b)),

[3] 8 processors with 8, 16, 32, and 64 blocks (Figure 7-5 (c)), and

[4] 16 processors with 16, 32, and 64 blocks (Figure 7-5 (d)).

For each experiment the run-time information is divided into computation and data structure building. The pattern is the same for all the runs: both computation and data structuring times decrease to a minimum at 32 blocks, and then increase for 64 blocks. Although this is an improvement over the naive method, there do exist further refinements. One communication refinement is to avoid the quick sequence of messages that are sent when a shell is communicated to processor that owns the “arrow-tip”. The next experiment address this problem by bundling the messages themselves.

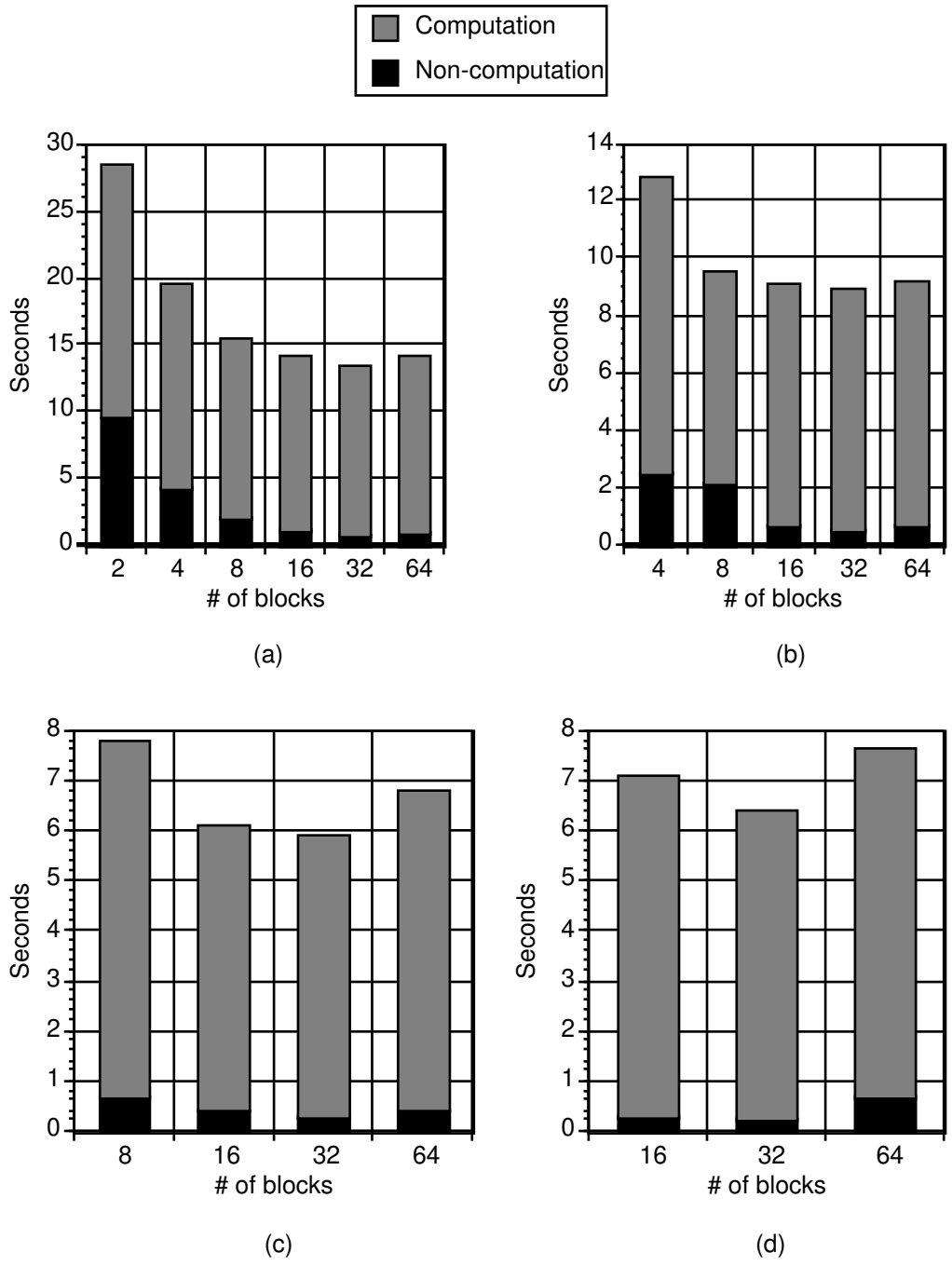


Figure 7-5. Run-times for block approach: (a) 2 processors, (b) 4 processors, (c) 8 processors, and (d) 16 processors.

7.3.3 Message Bundling

On distributed memory architectures, the time for starting up a message is considerably higher than the per-byte transfer time. The advantage of message bundling is that it

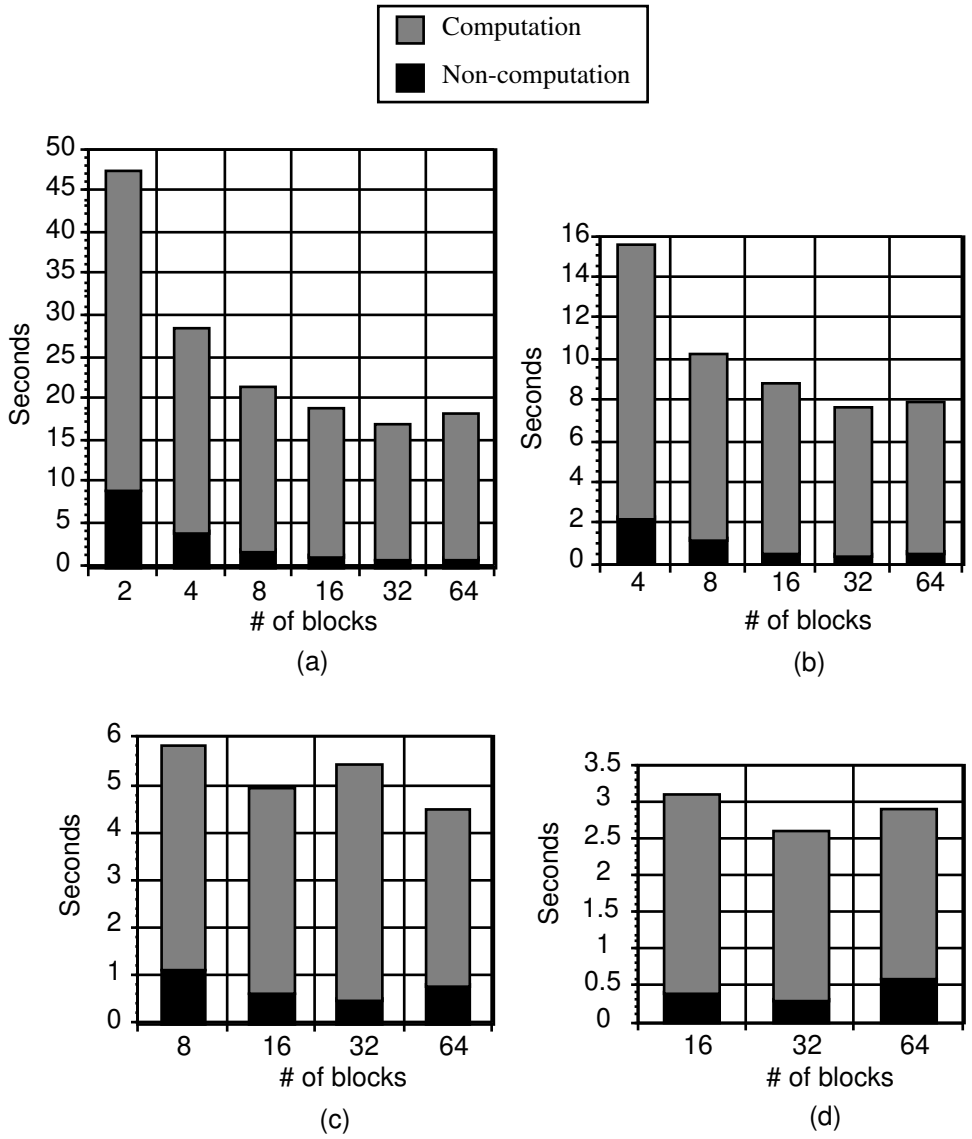


Figure 7-6. Run-times for message bundling approach: (a) 2 processors, (b) 4 processors, (c) 8 processors, and (d) 16 processors.

avoids the repeated overhead of starting up a sequence of messages by putting all the messages in a single message. In addition, we removed the restriction that the processors must wait before sending the bundled message. In order to quantify the advantages of this approach, we ran the same set of experiments as before:

[1] 2 processors with 2, 4, 8, 16, 32, and 64 blocks (Figure 7-6. (a)),

[2] 4 processors with 4, 8, 16, 32, and 64 blocks (Figure 7-6 (b)),

[3] 8 processors with 8, 16, 32, and 64 blocks (Figure 7-6 (c)), and

[4] 16 processors with 16, 32, and 64 blocks (Figure 7-6 (d)).

The results of this experiment indicate that message bundling is not always successful. For small numbers of processors (2 and 4) the overhead associated with message bundling seems to dominate the execution time. Although, for 4 processors with 32 blocks, it is more effective than the simple blocking strategy alone. With larger numbers of processors (8 and 16), message bundling is advantageous.

7.4 Discussion

We have performed a number of experiments in order to gauge the utility of the MaTRiX+$^+$ prototype. The experiments on the finite difference problem indicate that hierarchical type information can be used to great benefit. The MaTRiX+$^+$ system achieved speedups of up to a factor of 5. From the analysis of the overhead of the system it is clear that refinements of the system will yield even greater speedups.

The experiments on the power systems application investigated the ability of the MaTRiX+$^+$ system to exploit available parallelism. Although the system does automatically exploit parallelism, it is clear from the experimentation that communication issues greatly affect its performance. These communication problems go beyond just issues of latency: the size and frequency of messages must be taken into account. The message bundling approach indicated that for larger numbers of processors, good message management can improve performance.

In summary, the experiments have both demonstrated the utility of the hierarchical typing approach, and pointed out inefficiencies in the current MaTRiX+$^+$ prototype. Although the current system achieves speedups across a range of matrix types, it is clear that a highly optimized future version of the system will achieve considerably better performance.

Chapter 8

Contributions and Future Work

This chapter concludes the dissertation. It briefly summarizes the problem addressed by this research and the approach we have taken to it. Next, the major contributions of the research are reviewed. Then, a few of the possible interesting future directions for research are discussed.

8.1 Summary

Matrix computations arise in the implementation of almost all scientific and engineering applications. Due to the physical properties of these problems and the nature of the solution methods, the resulting matrices often have complicated structure. Although it is well known that this structure plays a crucial role in the efficiency of matrix computations, there exists little direct support in current programming systems for computations over the kinds of matrices that arise in modern applications. Also, the computations are often so large that parallelism is necessary to achieve acceptable execution times. The combination of structure and parallelism makes coding the numerical algorithms a tedious and error-prone task.

This dissertation presents formal and pragmatic models that provide powerful language and compiler capabilities for simplifying the task of constructing efficient parallel implementations for matrix computations. The central concept is a theory of hierarchical matrix computations. This theory provides a foundation for defining hierarchical matrix types and the matrix operations upon them. Hierarchical matrix types are an elegant means for describing both the computational and representational semantics of the matrices that arise in scientific applications. The definitions of matrix operations utilize this semantic information in efficiently generating hierarchically typed result matrices. It is this embedding of semantic in-

formation in the type system of the language that provides a basis for a powerful matrix computation system.

The MaTRiX+$^+$ environment is a prototype language and compiler implementation of the hierarchical matrix algebra; written in CC++, it has the important object-oriented benefits of clean interfaces, encapsulation, and easy extensibility. For the application engineer it presents a natural syntax in which to construct hierarchical matrix computations. This encourages the use of powerful abstractions for mapping scientific applications to hierarchically structured matrices. The principal language elements include: a number of simple type composition operators for building hierarchical matrix types, directives for mapping the matrices over distributed processors, and a function call mechanism for performing the matrix operations. The compiler transforms a source program's declarations of matrix structures into threaded, data-driven single-program multiple-data node programs in which matrix operations are recursively implemented through calls to the most appropriate and efficient matrix algebra library routines. Experimentation suggests that the specifications in MaTRiX+$^+$ are concise and result in performance that would be hard to achieve in all but hand tuned Fortran.

8.2 Contributions

This dissertation proposes a novel approach to scientific programming that makes both formal and practical contributions. The theory of hierarchical matrix computations provides a model for describing and exploiting matrix computation semantics. To the best of our knowledge, this is the first theory to provide a formal basis for language and compiler support for complicated, large-scale matrix computations. The MaTRiX+$^+$ prototype environment supports rapid specification of complicated matrix computations without sacrificing efficiency. The language interface provides a user with the conceptual capability to separate the semantics of the matrices from the algorithms that are performed over them. These hierarchical matrix semantics are exploited by the compiler to produce efficient sequential and parallel executions. This enables users to think about their applications in terms of their problem description rather than their solution in a Fortran-like language. The success attendant with use of semantic knowledge to obtain efficient programs for a significant domain suggests exploration of this approach in other application domains.

8.3 Future work

This research provides an important foundation for powerful language and compiler support for complicated, large-scale matrix computations. But as with any research, there always remains directions that have yet to be explored. We will briefly outline some of the more interesting directions for future research.

8.3.1 Theory extensions

The current theory presents an algebra for hierarchical matrices. It develops the concept of hierarchical type as a mechanism for capturing and exploiting the semantics of matri-

ces. It seems likely that this same hierarchical (or recursive) approach can be applied to other domains. Two possible domains are array computations and database applications. These areas are especially amenable to our approach because they both consist of operations over large, highly structured data representations.

8.3.2 Implementation extensions

The current MaTRiX+$^+$ implementation is a prototype that provides only the most immediate accessibility to the hierarchical matrix algebra. There exist numerous opportunities for refining and extending the system. Some of the superficial extensions include the addition of more types, operations, and algorithms. One major extension is the addition of explicit parallel structuring specifications to the system.

- The addition of explicit parallel constructs to MaTRiX+$^+$ would allow the exploitation of more useful parallelism. For instance, adding parallel directives in the matrix algorithms would enable the exploitation of writer-writer parallelism (Chapter 5). Also, it would support parallel operations, such as reductions, that are necessary for high-performance on applications such as the power systems example (Chapter 7).

Although these extensions would increase the utility of the MaTRiX+$^+$ prototype, we would gain substantially greater benefits by constructing a real compiler. A powerful static compiler for hierarchical matrix structures enables a number of useful optimizations.

- One important optimization is automatic data distribution. The experimentation demonstrated that it is difficult to determine effective data distributions. If we were to couple data distribution information with the matrix algorithms, it may be possible to choose useful distributions at compile-time.
- Another powerful compiler optimization is matrix structure transformation. This is the automatic process of converting a user's hierarchical type specification into one that yields better performance. One such transformations is structuring/un-structuring to obtain better granularity for the target architecture.

Although MaTRiX+$^+$ simplifies the specification of complicated matrix structures, it is doubtful that text-based description is the appropriate means for capturing matrix. A visual form is much more natural for this task.

- A graphical interface to MaTRiX+$^+$ has a number of advantages. It can speed-up specification by supporting menu-driven construction. Also, type checking can be performed interactively to avoid the compilation process for determining the legality of hierarchical matrix structures. In addition, a graphical "type simulator" could be used to analyze the utility of a given type description during a symbolic execution of a matrix operation. This would facilitate the determination of effective type descriptions.

8.4 A Final Word

Complexity it ubiquitous. Yet, often, there is some intangible order lying tantalizingly close to the surface. Computer science provides a number of techniques for dealing with complexity - one of the most powerful is recursion. Used as a tool, recursion can be used to capture complexity. But when used as an abstraction, it can sometimes expose simplicity.

Appendix A. Glossary of Hierarchical Matrix Types

Arrow	All elements off the main diagonal(s) and the "arrow tip" (the last row and column of the matrix) are of type **zero**.
Bidiagonal (lower/upper)	All elements either (above/below) the main diagonal or (below/above) the first (lower/upper) diagonal are of type **zero**.
Diagonal	All elements off the main diagonal are of type **zero**.
Hankel	A matrix that is symmetric across the north-east south-west axis.
Hessenberg (lower/upper)	All elements (above/below) first (lower/upper) diagonal are of type **zero**.
Identity	All elements along the main diagonal are of type **identity** and all other elements are of type **zero**.
Pentadiagonal	All elements above the second upper diagonal and below the second lower diagonal are of type **zero**.
Positive Definite	A square matrix for which pivoting during solving is unnecessary.
Strictly Diag. Dominant	A square matrix for which pivoting during solving is unnecessary.
Symmetric	For all (i,j), element (i,j) is the transpose of element (j,i).
Toeplitz	All elements on each diagonal are equivalent.
Triangular (lower/upper)	All elements (above/below) the diagonal are of type **zero**.
Unit	The main diagonal of the matrix is recursively of type **unit**.
Zero	All elements are of type **zero**.

Appendix B. Proofs of Lemmas and Theorems

B.1 Theorem 3-1

We will need some preliminary lemmas before we can prove Theorem 3-1 on page 21.

Lemma B-1 The application of an index, x, to a hierarchical matrix structure h in H results in either a scalar or a matrix structure of height at most height(h) - 1.

Proof We will prove Lemma B-1 in two parts. First, if h is in M then by Definition 3-1 h(x) will yield a scalar. Second, by Definition 3-11 we know that h is in $H^{height(h)}$ and therefore by Definition 3-6 that h(x) is in $H^{height(h)-1}$. Using Definition 3-11 again we find that height(h(x)) ≤ height(h) - 1 and therefore height(h(x)) < height(h).

□

Lemma B-2 Given a hierarchical matrix structure h in H, the length of a legal index sequence, x, in h is at most height(h) + 1.

Proof We will prove this lemma by induction.

Base case: (h in M) The only legal indices are the empty index (length 0) and indices that map to scalars (length 1). Therefore using Definition 3-11 we have 1 ≤ height(h) + 1 ≤ 0 + 1 ≤ 1.

Induction hypothesis: Assume Lemma B-2 holds for all matrix structures of up to height m.

Induction case: Given h in H such that height(h) = m + 1 prove that for any legal index sequence, x in h, that length(x) ≤ m + 2. We will break x into its *head*, an index sequence of length 1, and its *tail*. By Definition 3-6 h(head(x)) yields a matrix structure h´ of height at most m. Using the induction hypothesis, *tail* is of length at most m + 1. Therefore, length(x) ≤ m + 1 + 1 ≤ m + 2.

□

Theorem B-1 (Theorem 3-1 on page 21) *A hierarchical matrix structure is of finite height iff it contains no infinite sub-structure of 1-by-1 matrix spaces.*

Proof We will prove Theorem B-1 by proving an equivalent statement that a hierarchical matrix structure, h, is of finite height iff there exists no legal index sequence over h that has an infinite subsequence of (1,1) indices. We will prove this statement in two parts.

We will begin with the easy direction: if a matrix structure is of finite height then it has no legal index sequences with infinite subsequences of (1,1) indices. By Lemma B-2 we know that no finite matrix structure can have an index sequence of infinite length.

Now we prove the other direction: if there does not exist a legal index sequence that has an infinite number of (1,1) indices, then h is of finite height. We will prove this second part by contradiction by assuming that h is of infinite height. Take any of the longest index sequences in h, x. Consider the application of h(x) written as h(head(x))(tail(x)) (Definition 3-

12). Now h is an element of some H_m so we call its "size", s, which is equal to row-size(m)*colsize(m). Similarly, h(head(x)) is an element of some H_{m1} and therefore a "size" s_1. By Definition 3-3 and Definition 3-4 we know that s is greater than or equal to s_1. So if we continue the application of h(x) to its completion we obtain a non-increasing chain with an infinite number of elements. Since x does not contain an infinite number of (1,1) indices it must contain an infinite number of indices of the form (i,j) where either i or j is greater than 1. For each of the indices of this form the relation between the associated s_n and s_{n+1} is strictly greater than. (This is because the matrix space is partitioned in at least one direction and therefore into smaller spaces.) If we only consider these relations then we have an infinite decreasing chain. But this violates Def. 3-1 which guarantees that matrix spaces are of finite size. Therefore, we have contradicted our assumption thereby proving the theorem.

□

B.2 Theorem 3-2

Theorem B-2 (Theorem 3-2 on page 24) *Given a matrix structure h in H and some strong recursive property* $\overline{prop}$ *such that* $\overline{prop}(h)$ *holds then* $\overline{prop}(r_flat(h))$*also holds.*

Proof The proof of Theorem B-2 is very straightforward. From Definition 3-8 we know that r_flat(h) yields the matrix m which is the root matrix of h. Therefore we have only to prove that $\overline{prop}(h)$ implies $\overline{prop}(m)$. This is given directly by Definition 3-18.

□

There is also the obvious corollary to this theorem.

Corollary B-1 (Corollary 3-1 on page 24) *Given a matrix structure h in* H^2 *and some strong recursive property* $\overline{prop}$ *such that* $\overline{prop}(h)$ *holds then* $\overline{prop}(flat(h))$ *also holds.*

Proof We obtain this directly from the proof of Theorem 3-2 and that r_flat(h) equals flat(h) for elements of H^2.

□

B.3 Theorem 3-3

In order to prove Theorem 3-3 we first need a preliminary lemma.

Lemma B-3 Equipartitions (from a matrix structure h to h´ with g_partition(h,r,r)) maintain greater-than relations between elements of indices.

Proof We have to show that for all h´(i,j), if i is greater than j, then for all (i´,j´) where h(i´,j´) equals h´(i,j)(k,l) then i´ is greater than j´. Consider any legal index (i,j) where i is greater than j. The first element of that matrix structure (h´(i,j)(1,1)) will have index h(i´,j´) where i´ equals size(r,i-1)+1 and j´ equals size(r,j-1)+1 by Definition 3-9. Since the function size is monotonic we have i´ is greater than j´. Now we consider some index h´(i,j)(1,k) that

generates the largest j´ and the smallest i´ in this matrix structure. The resulting j´ index will be strictly less than the one indexed by h´(i,j+1) which is in turn less than or equal to h´(i,i). By transitivity and Definition 3-9, j´ is at most size(r,i-1). Therefore, i´ is always greater than j´.

□

Theorem B-3 (Theorem 3-3 on page 24) *Given a matrix structure h in H for which upper_triangular(h) holds. For any legal equipartitioning, h´ = g_partition(h,r,r), upper_triangular(h´) holds.*

Proof To prove that h´ is upper_triangular, we have to show that for all (i,j) where i greater than j, h´(i,j) is zero. By Lemma B-3 we know that for all indices h´(i,j)(k,l) where i is greater than j, the equivalent index, h(i´,j´), i´ is greater than j´. Since h is upper_triangular, for all (i´j´) where i´ is greater than j´ we have h(i´,j´) is zero. Therefore, by the definition of zero (Definition 3-14), h´(i,j) is zero and so h´ is upper_triangular.

□

B.4 Theorem 3-4

Theorem B-4 (Theorem 3-4 on page 25) *For all h in H if* $\overline{\text{upper_triangular}}$*(h) holds then upper_triangular(h) also holds.*

Proof We have to show that for all i greater than j, h(i,j) is zero. This comes trivially from Definition 3-20 through rearrangement of terms.

□

B.5 Theorem 3-5

Theorem B-5 (Theorem 3-5 on page 25) *For all h in H if lower_bidiagonal(h) holds then tridiagonal(h) also holds.*

Proof To prove Theorem B-5 we have to show that Definition 3-15 implies Definition 3-16. Using the anti-monotonicity of implication we obtain

$$\left((j > i+1) \wedge (i > j+1)\right) \text{ implies } \left((j > i) \wedge (i > j+1)\right)$$

which simplifies to

$$(j > i+1) \text{ implies } (j > i)$$

which is in turn trivially true.

□

B.6 Theorem 3-6

In order to prove the general theorem, we will first need a lemma that expresses the same concept but over block matrices (H^2).

Lemma B-4 For the expression A+B=C with A,B, and C in H^2, if Definition 3-23 holds recursively then that operation is equivalent to matrix addition.

Proof We will prove Lemma B-4 by showing that flat(A+B) = flat(A) + flat(B). We know by Definition 3-5 that there exist an a´ and b´ that are the roots of matrices A and B. Using the definition of partition (Definition 3-4) for all legal i,j,k,l we have flat(A(i,j)(k,l) + B(i,j)(k,l)) = flat(A(i,j)(k,l)) + flat(B(i,j)(k,l)). Again by Definition 3-4 we obtain c´(size(r_1,i-1)+k, size(c_1,j-1)+l) = a´(size(r_2,i-1)+k,size(c_2,i-1)+l) + b´(size(r_3,i-1)+k,size(c_3,i-1)+l). But by Definition 3-21 since A, B, are add compatible, we know that $r_1 = r_2 = r_3$ and $c_1 = c_2 = c_3$ and therefore the operation is equivalent to matrix addition.

□

Using this lemma we can now prove the general result.

Theorem B-6 (Theorem 3-6 on page 28) *For the expression A+B=C with A,B, and C in H, if Definition 3-23 holds recursively over A, B, and C, then that operation is equivalent to matrix addition.*

Proof We will prove Theorem 3-6 by induction.

Base case: the theorem holds for elements of M and H^2 by Definition 3-23 and Lemma B-4.

Induction hypothesis: the theorem holds for matrix structures of at most height m.

Induction case: consider structures A, B, and C of height m+1. We know that Definition 3-23 holds and therefore add_compatible(A,B,C) and for all (i,j) in the domain of C, C(i,j) = A(i,j) + B(i,j). But since any C(i,j) is of at most height m, this addition is equivalent to matrix addition by our induction hypothesis. Therefore we can replace each of these elements by its root resulting in A, B, and C all being structures of height 1. Using Lemma B-4 we have that addition over these structures is equivalent to matrix addition.

□

B.7 Theorem 3-7

As with the previous theorem, we will use a lemma to help with the proof.

*Lemma B-5 For the expression A*B=C with A,B, and C in H^2, if Definition 3-24 holds recursively over A,B, and C, then that operation is equivalent to matrix multiplication.*

Proof The proof that block matrix multiplication is valid is given in [13] on page 27.

□

The general version is as follows.

Theorem B-7 (Theorem 3-7 on page 28) *For the expression A*B=C with A,B, and C in H, if Definition 3-24 on page 28 holds recursively over A, B, and C, then that operation is equivalent to matrix multiplication.*

Proof We will prove Theorem B-7 by induction:

Base case: the theorem holds for elements of M and H^2 by Definition 3-24 and Lemma B-5.

Induction hypothesis: the theorem holds for matrix structures of at most height m.

Induction case: consider structures A, B, and C of height m+1. We know that Definition 3-24 holds and therefore, mult_compatible(A,B,C) and that for all (i,j) in the domain of C that C(i,j) = sum over i of A(i,k) * B(k,j). But since any A(i,k) and B(k,j) are of at most height m, this multiplication is equivalent to matrix multiplication by our induction hypothesis. Therefore we can replace each of these elements by its root resulting in A, B, and C all being structures of height 1. Using Lemma B-5 we have that addition over these structures is equivalent to matrix addition and therefore the operation is equivalent to matrix multiplication.

□

B.8 Theorem 3-8

Theorem B-8 (Theorem 3-8 on page 29) *Abstract block algorithms are hierarchically composable over compatible matrix structures*

Proof We will prove Theorem B-8 by induction.

Base case: consider some abstract block algorithm being applied to matrix structures, A_1, A_2, ...,A_n, in H^3. All the operations in this algorithm will be of the form: $op(A_1(i_1,j_1),A_2(i_2,j_2),...,A_n(i_n,j_n)$. Each of these indexing function will result in matrix structures in H^2. By Definition 3-25 and because the matrix structures are compatible, we know that some abstract block algorithm can correctly implement this operation. Therefore, the composition of abstract block algorithms can be used to correctly implement operations over matrix structures in H^3.

Induction hypothesis: assume that abstract block algorithms can be composed on matrix structures of up to height m in H^m.

Induction case: consider some abstract block algorithm being applied to matrix structures, A_1, A_2, ...,A_n, in H^{m+1}. All the operations in this algorithm will be of the form: $op(A_1(i_1,j_1),A_2(i_2,j_2),...,A_n(i_n,j_n)$. Each of these indexing function will result in matrix structures in H^m. By the induction hypothesis and because the matrix structures are compatible, we know that some abstract block algorithm can correctly implement this operation. Therefore, the composition of abstract block algorithms can be used to correctly implement operations over compatible matrix structures.

□

References

[1] E. Anderson et al. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Supercomputing*, pages 2-11. IEEE Computer Society Press, 1990.

[2] A.J.C. Bik, and H.A.G. Wijshoff. On Automatic Data Structure Selection and Code Generation for Sparse Computations. In *6th International Workshop on Languages and Compilers for Parallel Computing*, pages 57-75, eds. U. Banerjee et al, Portland, Oregon, August 1993.

[3] K.M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report, Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Technology, 1992.

[4] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - A Fortran Language Extension For Distributed Memory Multiprocessors. ICASE Report No. 91-72, NASA, September 1991.

[5] J.J. Dongarra et al. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, pages 1-17, 16(1990).

[6] J.J. Dongarra, R. Pozo, and D. Walker. LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra. LAPACK++ Manual. November 1993.

[7] J.J. Dongarra, R. van de Geijn, and D.W. Walker. Scalability Issues Affecting the Design of a Dense Linear Algebra Library. Submitted to *Journal of Parallel and Distributed Computing*. October 1993.

[8] J.J. Dongarra et al. LINPACK Users' Guide, SIAM, Philidelphia, PA, 1979.

[9] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1994.

[10] G. Fox et al. Fortran D Language Specification. Technical Report CS-TR90079, Rice University, March 1991.

[11] D. Gannon and J.K. Lee. Object Oriented Parallelism: pC++ Ideas and Experiments. In *1991 Japan Society for Parallel Processing*, pages 13-23, 1991.

[12] B.S. Garbow et al. Matrix Eigensystem Routines - EISPACK Guide Extension, vol. 51 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1977.

[13] G.H. Golub, and C. Van Loan. *Matrix Computations* 2nd edition. The Johns Hopkins University Press, Baltimore, 1989.

[14] A.S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Computer Science Report TR-91-07, University of Virginia, 1991.

[15] P. Hatcher, M. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones.

Data-Parallel Programming on MIMD Computers. In *IEEE Transactions on Parallel and Distributed Systems*, pages 377-383, Vol. 3, No. 2, July 1991.

[16] High Performance Fortran Language Specification. January 1993.

[17] K.E. Iverson. *A Programming Language*, John Wiley and Sons, New York, 1962.

[18] D. Ju, and W. Ching, Exploitation of APL Data Parallelism on a Shared-memory MIMD Machine, in Proceedings of 3rd ACM Syposium on Principles and Practice of Parallel Programming, 1991.

[19] D.P. Koester, S. Tanka, and G.C. Fox. A Parallel Gauss-Seidel Algorithm for Sparse Power System Matrices. Technical Report, SCCS-630, NPAC, April 1994.

[20] L. Mullin. A mathematics of arrays. Ph.D. dissertation, Syracuse University, 1988.

[21] J.T. Oden, and L. Demkowicz. h-p Adaptive Finite Element Methods in Computational Fluid Dynamics. In *Computer Methods in Applied Mechanics and Engineering*, pages 11-40, 89(1990).

[22] J.T. Oden, A. Patra, and Y. Feng. Domain Decomposition for Adaptive *hp* Finite Element Methods. In *Proceedings of the VII International Conference on Domain Decomposition*, State College, PA., October 1993.

[23] G. Po. Parallel h-p Methods for Coupled BE/FE Analysis of Structural Acoustics Problems. Ph.D. dissertation, The University of Texas at Austin, 1994.

[24] M. Rosing, R. Schnabel, and R. Weaver. The DINO Parallel Programming Language. Technical Report CU-CS-457-90, Department of Computer Science, University of Colorado, April 1990.

[25] H. Samet. The quadtree and related hierarchical data structures. *Computer Surveys* 16, 2 (June, 1984), 187-260.

[26] G.M. Skagestein. Rekursiv Unterteilte Matrizen Sowie Methoden Zur Erstellung Von Rechnerprogrammen Fur Ihre Verarbeitung. Ph.D. dissertation, Universitat Stuttgart, 1972.

[27] B.T. Smith et al. Matrix Eigensystem Routines - EISPACK Guide, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin 2 ed., 1976.

[28] D.S. Wise. Matrix Algorithms Using Quadtrees. Technical Report, Indiana University, CS TR354, June 1992.

[29] D.S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* 274, pages 134-153, ed. G. Kahn, Springer, Berlin, 1987.

Vita

Timothy Scott Collins was born in Sydney, Australia on February 17, 1966, the son of Geoffrey and Lynette Collins. After graduating from La Jolla High School in La Jolla, California in 1983, he enrolled at the University of California, Berkeley where he received the degree of Bachelor of Science in 1987. His major area of study was electrical engineering and computer science. During 1986 he spent a semester as a coop at Bell Communications Research in Red Bank, New Jersey performing research in computer aided design. He continued research at the Electronics Research Laboratory in Berkeley, California until he entered the Graduate School of The University of Texas funded by an Artificial Intelligence Laboratory Fellowship.

Permanent address: 14 Leeward Rd, Belvedere, California 94920.

This dissertation was typed by the author.