MaTRiX++: An Object-Oriented Environment for Parallel High-Performance Matrix Computations*

Tim Collins and James C. Browne

Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712

Abstract
The MaTRiX++ system is a high-performance matrix computation environment. It provides support for the task of constructing efficient implementations for the complicated matrix structures that arise in modern scientific and engineering applications. The foundation for this system is a hierarchical matrix algebra that defines hierarchical representations for structured matrices and recursive implementations for matrix operations. In this paper we discuss additions to the MaTRiX++ language interface and compilation model that will support applications written for distributed-memory parallel supercomputers. The language extensions provide methods for describing hierarchical data and computation distribution over the separate memories and processors of the target architecture. The compilation approach utilizes this distribution information in order to construct data driven single program multiple data (SPMD) processor programs. The high level of abstraction of the system encourages rapid prototyping without sacrificing efficiency.

1.0 Introduction
There has been relatively little attention paid to the development of a sound basis for the generation of efficient parallel executable codes for programs written in high level matrix languages. Historically, even sequential matrix languages have never achieved wide success in large-scale computations where the matrices have problem-specific structure because the compiled programs have never been acceptably efficient in execution. One important reason for this lack of efficiency is that the specifications for programs do not supply sufficient semantic information for a compiler to exploit the problem-specific structure of the matrices. Constructing efficient parallel, distributed-memory implementations of these matrix computations is even more demanding; it requires specification of a useful distribution of the computation and the matrix data themselves over the distributed processors.

1.1 A Hierarchical View
Applications of computational methods to realistic models of physical systems requires that the problem formulation span complex boundary conditions and regions of solution space with substantially different properties. Radiosity problems are a representative example. Modern methods for solving the partial differential equations of computational science and engineering are increasingly relying on locally adaptive approximations [1,2] which require solution of complex hierarchical matrix systems. One example matrix structure resulting from these methods is that proposed in a recent paper [33 on hp-adaptive finite element methods applied to computational fluid dynamics. The global matrix defining the problem is decomposed into domains. The matrices describing each domain each have a specific type such as zero, diagonal, block-triangular, etc. The matrices for each domain also has a further structure of submatrices. Solution of the global matrix by a uniform method will lead to excessive computational work while coding a formulation of the matrix in terms of a flat matrix of scalars is frighteningly complex. (There does exist some work in the area of restructuring compilers that attempts to deal with this problem [4].)

Similar examples can be found all across engineering and science. A natural representation of complex hierarchical matrix structure would make programs implementing adaptive methods and representing complex physical systems much more straightforward. The major issues are the definition of an appropriate set of abstractions in which to represent such computations and a compilation methodology which maps the abstract programs to efficient executables.
Efficient solution of such matrix systems requires efficient storage structures and algorithms for operating on each of the sub-matrices. Choices for representations and algorithms that are specific to a type structure can result in dramatically lower storage requirements and operation counts. Storage allocation and algorithm selection can be automated by a compiler if sufficient information about the data structures and operations is supplied. It has been demonstrated that the necessary information for a broad range of applications can be captured in a hierarchical matrix algebra [5]. This algebra consists of hierarchical type structures and a set of structured matrix operators which provide a framework for defining type-specific algorithms for matrix operations. The MaTRiX++ environment, written in C++, provides a language and compiler implementation of this hierarchical matrix algebra. Without the enabling technology of object-oriented languages, this prototype would have been extremely difficult to construct.

The MaTRiX++ prototype extends the usual concept of a matrix language by providing the user with language constructs for declaratively describing the hierarchical structure of application-specific matrices. This additional semantic information allows the compiler to resolve the computation into type-specific algorithms for the sub-matrices through automatic choices of efficient storage structures and base level routines. We have chosen LAPACK++ [7] as the source of these primitive routines. LAPACK++ is an attractive choice because it has demonstrated that a numerical package written in C++ can achieve Fortran-like execution speed.

1.2 The Problem and Approach

The problem addressed in this paper is that of extending the MaTRiX++ environment to provide language and compiler support for matrix computations on distributed-memory parallel supercomputers.

The additional specifications to define a parallel execution model for MaTRiX++ programs are: (i) specification of a hierarchical logical mesh of processors, (ii) distribution of the hierarchical matrices over the hierarchical mesh of processors and (iii) additional specification of the processors as being "readers", "writers", or "spectators" of their data during a given computation step of an SPMD execution model.

There are several factors of the implementation of this model of execution which may not be obvious. The hierarchical mesh of processors is in fact a mesh of logical processes where the hierarchical decomposition is chosen to conform to the hierarchy of the matrices as far as is possible. The logical mesh may be mapped to any physical topology of processors. (From now on we will use the term processor to denote a process.) Distribution of matrices is accomplished at run-time through the instantiation of the hierarchical matrix objects. Once a hierarchical mesh is declared then the specific instantiation of the execution model for a given computation can be specified declaratively by adding a distribution type specification to the type definitions of the hierarchical matrices. The compiler is then able to implement the data distribution, synchronization, and communication operations as it derives the logical structure of the hierarchical matrices.

The contributions of this work are simple language and compiler models which provide both a high level of abstraction for describing complicated matrix computations and efficient implementations on distributed memory supercomputers. We believe that MaTRiX++ will provide a useful rapid prototyping capability for matrix applications.

1.3 Paper Contents

The remainder of the paper is structured as follows. The next three sections present a brief summary of the nature of matrix structure and how it can be exploited; Section 2 describes the kinds of matrix structures that we are interested in capturing; Section 3 covers elements of the hierarchical matrix algebra as a formalism for describing these structures; Section 4 presents the current MaTRiX++ system: its language interface and its compilation model. The next two sections present the proposed extensions to the MaTRiX++ system; Section 5 describes the extensions to the MaTRiX++ language interface that support distributed memory implementations; Section 6 defines a compilation strategy that results in data driven single program multiple data processor programs. The final sections provide brief coverage of the implementation, the related work, and the future extensions.

2.0 Structure: Its Sources and Exploitation

A matrix is structured if there exist constraints on the organization and nature of its elements. 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 "acoustic" matrix shown in figure 1. This matrix, although clearly structured, is not directly captured by any well-known matrix type.
2.1 Benefits of Structure Exploitation

Exploiting matrix structure is important since it can potentially be used to reduce both the number of data elements that need to be represented and the amount of computation that needs to be performed. Consider the representational and computational advantages of exploiting the structure of our example. Of the over one million data elements, less than 15% of them contain non-zero values. Similarly, avoiding operations over zeroes can yield a tremendous computational advantage since the algorithmic complexity of the matrix operations is at least linear in the size of the data.

3.0 The Hierarchical Matrix Algebra

The hierarchical matrix algebra defines abstractions for describing both the structure of matrices and the methods for performing matrix operations over them. First it defines hierarchical type structures that capture the kinds of deeply structured matrices that arise in modern scientific applications. Second it defines recursive matrix operators over the resulting hierarchical structures. We will briefly discuss some elements of the hierarchical matrix algebra. A complete, formal description is given in [5].

3.1 Hierarchical Type Structures

Historically, matrix properties such as tridiagonal, symmetric, identity, etc., have been defined to capture constraints on the organization and the nature of elements of a matrix. These typical properties are sufficient for describing the nature of matrices in some applications, but are inadequate for many modern applications. In the hierarchical matrix algebra, the standard matrix properties are extended to hold over matrices of structures in which elements may have properties themselves. The hierarchical extensions are defined generally so that they will work for matrices in areas such as radiosity [9] and computational fluid dynamics [3] that can be viewed as having two or more levels of structure. The elegance of the hierarchical approach is that the small number of known matrix properties can now be used to capture a vast number of matrix types.

Given these extensions, one view of the example in figure 1 is a hierarchical type structure composed of four subtypes, A, B, C, and D, in which type A has tridiagonal structure. One thing to note is that this type description, the one that would typically be captured in a hand coded implementation, represents many zero entries. In fact, more than 70% of the entries are zero. Hierarchical type structures provide the capability to capture more of this additional structure.
4.0 Current MaTRiX++ Implementation

The current MaTRiX++ implementation is a C++ package that provides mechanisms for both describing the kinds of hierarchical matrix structures that arise in modern scientific applications and performing matrix operations over them. MaTRiX++ separates the construction of structures, which are types, and the generation of instances, which are specific matrix objects that will contain values. The actual matrix operations are accessed through simple interfaces that mask all implementation detail. The compilation model relies heavily on the C++ runtime environment in order to determine appropriate type-specific implementations of matrix operations. We will briefly discuss the elements of the MaTRiX++ system that will be extended in the distributed-memory implementation. We will begin with the language interface.

4.1 MaTRiX++ Language Interface

MaTRiX++ provides language mechanisms for describing hierarchical matrix types, for instantiating them as matrix objects, and for performing matrix operations over these objects. We will give a brief overview of the pertinent features of these mechanisms; a more detailed description can be found in [6].

4.1.1 Matrix Types

Matrix types are semantic representations for hierarchically structured matrices. The interface provides mechanisms for describing typed matrices and composing them into hierarchical structures. There are two kinds of typed matrices: base matrix structures, matrices of scalar data, and hierarchical matrix structures, matrices of structures. The syntax for declaring a named base matrix structure is as follows:

```
MTRX_Type name_type(row,int,col,int,property,matrix,structure);
```

The matrix properties include a majority of the familiar kinds of matrices that arise in scientific applications. They are organized into the hierarchies shown in figure 3. A legal matrix property descriptor consists of at most one property chosen from each set. The BaseType is one of the common base data types: int, float, double, or complex. In the event that no BaseType parameter is given, a hierarchical matrix structure is constructed.

For hierarchical matrix types, it is 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 type setting function, assigns the same matrix type to all elements of a given selected region. Once a matrix type is used to define a region of another matrix type its structure may no longer be changed.

In order to describe the type of the matrix in figure 2, we must describe the types of regions A, B, C, and D and then compose them. The type of the composed matrix is described in the following code fragment.

```
MTRX_Type AcousticType(2,2);
MTRX_Element Element;
```

4.1.2 Matrix Objects

Matrix objects are instances of matrix types. Unlike a 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, describing references to matrix objects, and inputting and outputting data values.

Matrix objects are defined from matrix types through a simple syntax:
A matrix object derived from a base matrix structure is called a base matrix object. One derived from a hierarchical matrix structure is called a hierarchical matrix object. It is sometimes useful to have a matrix object that refers to some rectangular sub-matrix of another matrix object. Such entities are called reference objects. They have the property that operations performed on them change the values of their associated matrix object. Reference objects are defined as follows:

\[ \text{MTRX\_Object \ name1\_object(name2\_type).} \]

Once initialized, a matrix object or reference object can input and output values of the underlying matrix.

\[ \text{name1\_object.ref(name2\_object, region\_selector);} \]

4.1.3 Matrix Operations

The current MaTRiX++ prototype provides the set of matrix operations necessary for solving linear systems of equations. These operations include addition, general multiplication, solve, LU factorization, Cholesky factorization, and triangular solve. An example of the syntax for these matrix operations follows.

\[ \text{MTRX\_Add(in1\_object, in2\_object, out\_object);} \]

The input objects to these operations must be structured. Whereas the output (result) object may not necessarily be structured. Each operation can generate the appropriate structure for the result object based on type propagation functions defined in the hierarchical matrix algebra.

4.2 MaTRiX++ Compilation Model

The current MaTRiX++ implementation is a package written in C++. Therefore, MaTRiX++ programs are just compiled as C++ programs. MaTRiX++ programs rely heavily on dynamic binding for the determination of appropriate implementations for matrix operations. The dispatch mechanism utilizes matrix type information that annotates matrix objects for composing type-specific matrix implementations. In this section we will briefly summarize the nature of the algorithms that implement matrix operations, and the current mechanism for dispatching them at runtime.

4.2.1 Matrix Algorithms

Matrix algorithms 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. Most libraries do not implement all the needed algorithms. Therefore, mechanisms must be implemented for handling the unsupported combinations of input types. MaTRiX++ utilizes type coercion for these unsupported type combinations.

A matrix algorithm is defined as a C++ class. The class has member functions that return matrix type information about their input and output operands. In addition there is a special member function called apply. This function contains the code that implements the algorithm. Algorithm are made accessible by creating an instance of their class and registering it with the associated matrix operation.

4.2.2 Algorithm Dispatch

The C++ runtime environment is used for dispatching appropriate algorithms matrix operation implementations. The following items describe the major steps of the dispatch routine:

1. Check for compatibility of matrix operands.
2. If the operation is not to be performed in place then allocate result storage.
3. Determine the most type-specific algorithm. (Type coercion may be used.)
4. Call the apply member function of the chosen algorithm.
5. Update the type information of the result object.

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 faster. (Our approach requires the constant cost of hashing a key in order to obtain an offset in a function dispatch table.). The ability to code such a mechanism is a quintessential advantage of the object-oriented approach.
These interface extensions require little modification of existing specifications written in MaTRiX++.

5.1 Hierarchical Processor Meshes

Hierarchical processor meshes are descriptions of distributed-memory architectures that provide necessary information for both the distribution of matrix object data across the memories of the processors, and the sharing of structures by groups of processors. The user's task 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 p/m by q/n. Partitioning may be repeated until the mesh size is 1 by 1.

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

5.2 Distribution Directives

The next step is to describe how hierarchical matrix structures themselves are to be distributed over the processors. The mechanisms we use are distribution directives that are added to the parameter list of a matrix type definition. Each directive tells how the elements of the instantiated hierarchical matrix type are to be distributed relative to a local description of the processor mesh. 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 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. Distributions are used in matrix type definitions as follows:

- **MTRX_BLOCK**: The directive MTRX_BLOCK distributes the elements of a hierarchical matrix object over the processor mesh in the same way as a partition. That is, given an m by n matrix object A and \( \psi \), element \( A[i,j] \) of the matrix object is annotated with following element of the mesh descriptor.

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

- **MTRX_ALL**: The directive 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. The directive MTRX_ALL implies replication of a matrix object rather then its distribution.

The default distribution, if no directive is specified, is MTRX_CYCLIC. This promotes the exploitation of parallelism whenever possible. Figure 4 shows examples of each of the distributions.
Figure 5. The local representations on processor (1,1) (c) and processor (2,2) (d) given a matrix (a) and a processor mesh (b).

6.0 MaTRiX++ Parallel Execution Model

In this section we discuss the extensions to the MaTRiX++ execution model for parallel execution on distributed-memory architectures. Our target model is single program multiple data (SPMD) processor programs with explicit message passing for communicating non-local objects. Our two concerns are 1) how matrix objects are constructed and 2) how matrix operations are performed.

6.1 Matrix Object Construction

Matrix objects are instantiated matrix types that encode necessary semantic information in the form of annotations. The objects themselves are represented by tree data structures where nodes are matrix object descriptors and edges point from matrix object elements to matrix objects that refine the description of the element. Based on the distribution directive, each element of a matrix object is annotated with an appropriate element of the processor mesh descriptor. For a distributed-memory representation we utilize this annotation to determine the matrix object representations that each processor constructs.

A processor constructs all matrix objects that are annotated with its processor number (name). Figure 5 illustrates the matrix objects constructed by a set of processors, given an annotated matrix type. Matrix objects that are results of matrix operations are special in that they may have a structure and distribution which is determined at runtime. The sources of the structure and distribution information for these matrix objects are rules associated with the matrix algorithms. A matrix algorithm, if needed, can provide a distribution directive for its result object.

One important result of the representation rule is that all processors (described in the mesh initialization) 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.

6.2 Matrix Operation Execution

The MaTRiX++ runtime system uses a data-driven SPMD model of execution. In this model of execution, all processors execute the same code, yet different processors take different data-dependent paths through the program. A processor performs a matrix operation if its name annotates any of the participating matrix objects. The justification for this rule is that if a processor owns no part of the matrix objects, then the result of the operation can have no effect on its local name space. Since the programs execute on distributed memory architectures, it may be necessary to communicate non-local data. In this section we will discuss the role of communication, the models of communication, and the execution strategy.

6.2.1 Matrix Operation Notation

In this section we will use the notation \( M : (I_1, I_2, ..., I_n) \to R \) to describe a matrix operation \( M \) that takes \( n \) input objects and produces one result object. In addition we define \( \Psi = \Psi(I_1) \cup \Psi(I_2) \cup ... \cup \Psi(I_n) \).
Reader

if (base element operation)
    send inputs to $\Psi_R$
else
    spawn thread
    send inputs to $\Psi_U$
    receive inputs from $\Psi_I$
    receive result from $\Psi_R$
perform matrix operation

Writer

wait until all threads finish
if (base element operation)
    receive inputs to $\Psi_I$
else
    perform matrix operation

Spectator
	nop

Figure 6. Pseudo code for processor roles

$\Psi_R = \Psi(R)$, and $\Psi_U = \Psi_I \cup \Psi_R$ where $\Psi_I$ is the set of processors that own inputs to the operation, $\Psi_R$ is the set of processors that own the result of the operation, and $\Psi_U$ is the total set of processors involved in the matrix operation.

6.2.2 Role of Communication

MaTRiX+ programs often require the communication 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 $\Psi_I$ to send to the processors in $\Psi_R$ 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. Instead of communicating a whole matrix object, we only send the next level of description of the object. 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 $\Psi_U$. The reason that the result object must be communicated is so the processors in $\Psi_I$ 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 $\Psi_I$ to $\Psi_R$. 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.

6.2.3 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 is straightforward. Given two matrix objects $M_{01}$ and $M_{02}$, there are only four different possible relationships between the processors of $\Psi(M_{01})$ and $\Psi(M_{02})$:

\begin{align*}
\Psi(M_{01}) = \Psi(M_{02}) & \quad (1) \\
\Psi(M_{01}) \supset \Psi(M_{02}) & \quad (2) \\
\Psi(M_{01}) \supseteq \Psi(M_{02}) & \quad (3) \\
\Psi(M_{01}) \cap \Psi(M_{02}) = \emptyset & \quad (4)
\end{align*}

Here we are considering the communication of $M_{01}$ to $\Psi(M_{02})$. In the cases of relationships (1) and (2), since $\Psi(M_{02})$ already has a copy of $M_{01}$, no communication is necessary. For relationships (3) and (4), $M_{01}$ must be communicated. Since all processors in $\Psi(M_{02})$ have a copy of the matrix object, any one of them could potentially send its description to the processors in $\Psi(M_{01})$. It is desirable for the sake of communication speed to use broadcasts in lieu of point-to-point communication whenever possible.

6.2.4 Execution Strategy

As mentioned earlier, the model of execution is data-driven SPMD where matrix object annotations determine processor execution. But we still must decide how the node programs should guarantee a meaningful and successful execution order. Our approach is very simple. We do something analogous to a multiple reader single writer
protocol. We assign a *role* to each processor for each matrix operation in the program. We call these roles *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 $P_{i,j} \notin \Psi_j$, a writer if $P_{i,j} \in \Psi_j$, and a spectator if $P_{i,j} \in \Psi_j$. The manner in which a processor executes a matrix operation depends on its role, and whether the operation is a hierarchical element operation or a base element operation. We impose the following ordering rule 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 6 provides the pseudo-code for the events associated with a matrix operation role.

### 7.0 Implementation and Validation

The current MaTRiX++ prototype is written in C++. It is presently being tested on an IBM RS/6000 workstation on two scientific applications: an acoustics problem and a computational fluid dynamics (CFD) problem. The coding of the distributed-memory implementation of MaTRiX++ is almost complete. It will be tested on a network of RS/6000s before being ported to some distributed memory parallel supercomputers such as the Intel iPSC/860 and Paragon. We plan to compare the execution speeds of MaTRiX++ and Fortran based implementations of the acoustic, CFD, and other problems.

There exist a number of parallel dialects of C++ including pC++ [11], Mentat [12], and CC++ [13]. For our model of computation, the most appropriate of these languages is CC++. It provides an SPMD model of computation that supports the spawning of threads, global variables, and single-assignment variables. These abstractions allow an implementation of the distributed-memory version of MaTRiX++.

Recoding the current specifications for the new version of MaTRiX++ is a minor task. It only requires the addition of a mesh descriptor and distribution directives for the matrix types. We will then compare runtimes of the specifications on single and multiple nodes in order to measure speedup. Next we will construct in MaTRiX++ the “best” parallel specification that we can. If its execution time is comparable to the Fortran-based versions that already exist, we will consider the prototype to be a success.

### 8.0 Software Engineering Issues: Abstractions and Component Reuse

The fundamental goal of this research is to simplify the development of efficiently executing programs for computations over matrices with complex hierarchical structure. The approach is to (i) define an appropriate set of abstractions in which to represent these programs, (ii) define the semantic content necessary to support compilation to efficient executables, and (iii) define and implement a translation system which instantiates these concepts. This paper carries this approach into parallel execution environments.

Component reuse, one of the “Holy Grails” of software engineering is a by-product of compilation of executables from high level abstractions. The compiler actually only generates the macro-structure of the computation and the flow of control through the structure of the computation. All of the computations are done by calls upon the components provided in the LAPACK++ linear algebra.

Of course, this example of component reuse applies only in the semantic domain of computations defined on hierarchical matrices. But the concept is broadly applicable. Component reuse is enabled by use of appropriate abstractions for the representation of the problem.

### 9.0 Related Work

A number of languages provide mechanisms for supporting matrix computations on distributed-memory architectures. Although many of these languages, including HPF [14], Vienna Fortran [15], Fortran D [16], DINO [17], pC++ [11], CC++ [13], and ScaLAPACK [9], provide mechanisms for describing data and computation distribution, these mechanisms are not adequate for many of the types of matrix computations that arise in modern scientific applications. Matrices of this sort are more amenable to the hierarchical approach of MaTRiX++.

In MaTRiX++ we have adopted approaches used in current languages. The block and cyclic data distributions have been used in many existing systems [10,11,14-16]. But none of them use hierarchical data distributions. Also, we use an SPMD model of computation as do many of the current approaches [11,16,18]. But instead of exploiting data-parallelism as they do, MaTRiX++ exploits functional parallelism among coarser grain operations.

MaTRiX++ can be viewed as an extending the matrix type system of matrix packages such as LAPACK [8]. Our approach differs from their distributed-memory matrix package, ScaLAPACK, in the way that we exploit parallelism. ScaLAPACK exploits parallelism within the level 3 BLAS [19]. Our recursive formulations of matrix operations allow us to exploit parallelism among calls to level 3 BLAS.

### 10.0 Future Extensions

The proposed extensions to the MaTRiX++ system provide users with language mechanisms for expressing information that is necessary for deriving distributed-memory implementations of complicated matrix computations.
These mechanisms support the hierarchical decomposition of matrix objects across the separate memories of the architecture. Since the domain of matrix computations is so restrictive, it is possible that these distribution decisions could be equally well made by a compiler.

A compiler could have the capability of dynamically restructuring and redistributing matrix objects in order to yield more efficient computations. These decisions could be based on granularity and locality consideration. For example, a user’s matrix type descriptions may result in yielding more efficient computations. These decisions could be equally well made by a compiler.

so restrictive, it is possible that these distribution decisions could be equally well made by a compiler. These mechanisms support the hierarchical decomposition of matrix objects across the separate memories of the architecture. Since the domain of matrix computations is so restrictive, it is possible that these distribution decisions could be equally well made by a compiler.

A compiler could have the capability of dynamically restructuring and redistributing matrix objects in order to yield more efficient computations. These decisions could be based on granularity and locality consideration. For example, a user’s matrix type descriptions may result in yielding more efficient computations. These decisions could be equally well made by a compiler.

Computational transformations are only one aspect of the solution of a problem. Clearly advantage can be taken of type information to invoke type specific routines for I/O and for filtering, merging and transport of structured matrices. These additional optimizations will be considered for later versions of MaTRiX++.

Acknowledgments

We would like to thank an anonymous referee for his thoughtful comments which have led to substantial improvements in the presentation of the material. In addition we would like to thank Robert van de Geijn and Raju Pandey for technical assistance and Donovan Kolbly, Mark Johnstone, and Sheetal Kakkad for their C++ knowledge.

References