The proposed project will pursue the theory and practice of algorithms for linear algebra operations, and their implementations, when the matrices and/or vectors are stored recursively by blocks, as hierarchical matrices (Hyper-Matrices). The goal is to formalize abstractions for such data structures, to develop Application Programming Interfaces (APIs) that allow the practical development of entire dense and sparse linear algebra libraries specialized for these new data structures, and to implement proof-of-concept libraries. The proposed infrastructure will also be exploited for the transparent scheduling and optimization of operations on submatrices targeting SMP and multi-core architectures.

With the advent of processors with complex multi-level memories and architectures, a number of projects have started to re-examine how matrices should be stored in memory (a thorough review of these projects can be found in a recent SIAM Review paper by Elmroth et al.). The primary goal is to improve performance of basic linear algebra kernels like the level-3 Basic Linear Algebra Subprograms (BLAS), a set of matrix-matrix operations that perform $O(n^3)$ computations on $O(n^2)$ data, as well as higher level linear algebra libraries like LAPACK. By storing blocks at different levels of granularity packed in memory, costly memory-to-memory copies and/or transpositions can be avoided. These copies are currently required to provide contiguous access to memory and/or to reduce cache and TLB misses. While conceptually the previously proposed solutions are often simple and elegant, complicated indexing continues to prevent general acceptance.

The new insight that underlies the proposed project is simple but powerful: Storage by recursive blocks is typically explained as a tree structure with submatrices that are stored contiguously at the leaves, and inductively as blocks of submatrices at each other level of the tree. Thus, a data structure, Hyper-Matrix, that reflects this tree is the most natural way of expressing hierarchical matrices. Similarly, algorithms over these trees are expressed as recursive algorithms. An API for implementing recursive algorithms that obey this tree provides a natural solution.

The proposed research is particularly timely given the emergence of IBM’s Cell processor and other multi-core architectures that have memory local to each core. These architectures require hand-managed data moves. Hyper-Matrices can store data contiguously at the leaves, which simplifies these data moves, and the proposed API simplifies the programming effort. Moreover, the leaf submatrices and operations on them become units of data and computation for scheduling purposes, allowing techniques used for scheduling operations on super-scalar architectures to be transparently employed. Thus, the proposed work meets a critical need for new programming techniques that support such new architectures.

The team assembled at UT-Austin has unique qualifications for the proposed project: To make a meaningful assessment of the benefits of storage by blocks, it is essential that the best implementations that target traditional storage be compared to the best implementations that utilize hierarchical storage. Mr. Kazushige Goto (CoPI) rejoined UT-Austin in Fall 2004 and has implemented the fastest BLAS (GotoBLAS) for all widely used sequential architectures. The FLAME project, headed by Prof. van de Geijn (PI), has developed the most advanced approach to the development of linear algebra libraries, including derivation and implementation. Both these projects have yielded libraries that are available under Open Source licenses.

The intellectual merit of the proposed work lies with the demonstration that by raising the level of abstraction at which one thinks of hierarchical matrices, and the way they are stored, the code which implements algorithms that perform operations on such matrices is greatly simplified. This is a key step that will motivate the scientific computing community to embrace such a new approach to coding matrix operations. It also lies with the empirical and theoretical analyses of how algorithms and architectures interact when matrices are stored hierarchically. A final key contribution is to show superior performance, which is always the primary motivation in this community.

The broader impact lies with the tools, in form of libraries, that will be made available to the computational science community, and the pedagogical benefits of a simplified approach.

In the body of this proposal and letters of support from industry and government labs, evidence is presented that the proposed work is (1) of scientific importance, (2) based on sound preliminary results, (3) achievable, (4) of benefit to the scientific computing community, and (5) of pedagogical value.
1 Objective

The objective of the proposed project is to create a unified approach to the storage and manipulation of hierarchically stored matrices (Hyper-Matrices). It will be shown that storing Hyper-Matrices in a hierarchical datastructure enables solutions to a number of challenges that arise from the introduction of complex multi-level memories and multi-core architectures, as well as the hierarchy in successively substructured applications that lead to sparse linear systems. These include: (1) A generalization of storage schemes that use space-filling curves like the Morton ordering; (2) The high-performance implementation of dense linear algebra algorithms through data locality; (3) The compact storage of matrices with special structure (e.g., symmetry or triangular structure); (4) The typing (e.g., with structural or numerical properties) of submatrices; (5) Storage and manipulation of sparse linear systems that arise from hierarchically decomposed domains; (6) The optimization of scheduling of tasks on multiple threads via a near-transparent (automatic) mechanism similar to scheduling algorithms employed by super-scalar processors; and (7) The simplification of data movement on many-core architectures by presenting natural units of communication. The goal is to develop a theory and infrastructure that addresses these (often related) issues in a unified framework. We expect the results to influence future algorithms/programming practices, compilers, and architectures.

2 Relation of current proposal to a previously submitted proposal

This proposal is a refinement of a proposal to NSF’s “S and T High-End Computing” program, submitted in July 2004. That proposal received a “Good”, three “Very Good”, and an “Excellent” review. It was rated “Recommended Plus” but was not funded. That original proposal different from the current one in that it only addressed items (1)–(3) in the above list of opportunities. In addition, since the submission of that proposal, prototype implementations of the API (the FLASH, for Formal Linear Algebra Scalable Hyper-Matrix, API) were successfully used in settings related to opportunity (1)–(2) in a graduate class on high-performance computing and are being used by a female Ph.D. student at the University of Jaume I, Spain, for a dissertation on the out-of-core implementation of algorithms related to control theory. Subsequent to the original submission, CoPI Kazushige Goto has returned from Japan to work full-time at the Texas Advanced Computing Center of UT-Austin and is now a close collaborator and CoPI on the FLAME project, making the success of the proposed project even more assured.

3 Technical Rationale

The technical rationale behind the proposed project lies with the fact that (1) the development of linear algebra libraries is constrained by the practice of indexing directly into the buffers that store matrices; and (2) the increasing need for data locality as architectures, like IBM’s Cell processor, and memories have become more complex and more hierarchical. Here we discuss a number of closely related topics and how Hyper-Matrices represent an opportunity for furthering the state-of-the-art in the field.

Generalization of storage based on space-filling curves Considerable attention has been focused on the virtues of space-filling curves, like the Morton-ordering, which map multi-dimensional space (in this case two-dimensional indexing of arrays) into one-dimensional space (in this case linear memory) in such a way that nearness in the multi-dimensional space almost always translates to nearness in the one-dimensional space [21]. The problem with the conventional approach is that it inherently creates the hierarchy as a quad-tree. This may or may not match the relative sizes of the layers of a multi-level memory. An additional problem arises when dealing with matrix sizes that aren’t exact multiples of the various block sizes: how to map the “fringe” to memory is a messy proposition [45].

Our approach is fundamentally different. By viewing matrices hierarchically as matrices of matrices, the inherent dependence on powers of two disappears and dealing with the fringe becomes simple, as detailed later in this proposal.

High performance through data locality Hierarchical storage of matrices in general and storage by blocks in particular has gained a loyal following within the dense linear algebra libraries community (see [19] for a survey). The main purpose is to improve performance by improving data locality. Unfortunately, demonstrating performance improvements has been illusive, even for what appear to be natural candidate operations, like the level-3 BLAS. The first reason is that the best implementations for the level-3 BLAS, like the GotoBLAS [23], amortize the packing of data into contiguous memory so well that the benefits of storage by blocks disappear. A second, equally important, reason is that coding algorithms over matrices that are stored by blocks is hugely complicated by the translation of the
indexing into the “flat” matrix into the indexing into the hierarchically stored matrix. Much research has gone into how to use compilers to make the translation or how to use object-oriented programs [45, 48]. The purpose of these efforts is to allow the programmer, as much as possible, to continue to view the matrix as a “flat” matrix, hiding the hierarchy entirely.

Our approach is fundamentally different. Rather than shunning the hierarchical nature of the matrix, we embrace it. The translation of the indexing into the flat matrix to the hierarchical storage is avoided entirely by writing algorithms in a manner that obeys the hierarchical nature of the matrices. This solves the second problem mentioned above.

The first problem is solved by the involvement of Kazushige Goto in the project, as CoPI. Previous efforts have had a hard time demonstrating performance because most computation was being cast in terms of a matrix-matrix multiplication with the leaf nodes. This computation was performed by a call to the `DGEMM` BLAS routine, which massaged the data into a format that allows high performance. Since the same block is often used in many of these leaf-level multiplications, data is massaged multiple times, which impedes performance. Since our research has access to the low-level operations that massage the data and perform the subsequent multiplications, these overheads can be avoided and a fair performance comparison can be performed.

The approach also naturally supports the explicit management of units of data in complex architectures with explicitly managed memory hierarchies, like IBM’s Cell processor and the TRIPS processor [12].

**Compact storage** A closely related issue deals with the compact storage of matrices with special structure, e.g., triangular and symmetric matrices. Complex indexing schemes for mapping such matrices to memory so that only half the matrix is stored have been proposed [4, 35]. The problem is that special computational kernels, like the BLAS, have to be implemented that can utilize these new storage methods. The implementation is both tedious and overwhelming if entire libraries like the BLAS and LAPACK are to be targeted.

Our approach is fundamentally different. We recognize that when a triangular or symmetric matrix is viewed as a matrix of matrices, submatrices that are not referenced need not be stored. Branches of the hierarchy associated with these submatrices simply become “null” branches. Computation on blocks can utilize low-level kernels designed for matrices stored by blocks.

**Sparse direct** Traditional sparse linear system solvers take sparse matrices expressed in a format that indicate the nonzero entries in the matrix, perform an ordering on the rows and columns (degrees of freedom) in order to minimize fill-in, perform a symbolic factorization to determine space required to accommodate fill-in, and finally factor the matrix [15]. In order to attain high performance, super-nodes that represent dense subproblems must be identified from the sparsity structure.

Our approach is fundamentally different. Many sparse linear systems arise from hierarchically partitioned domains (either via partitioning packages like Metis [39, 41] or due to the natural structure that occurs when domains are refined). This partitioning naturally exposes hierarchy and super-nodes in the matrix. Storing the matrix as a Hyper-Matrix allows the computation over the sparse matrix to obey this hierarchy and to exploit the dense subproblems that occur. Also, since branches in the hierarchy can be dynamically added, fill-in can occur as the factorization proceeds rather than requiring the symbolic factorization as a preprocessing step.

**Typing** Matrices are often viewed as matrices of matrices, where different submatrices have different properties. For example, in the modeling of the electro-magnetics of an airplane, a Finite Element Method that models the interior structure is often coupled with a Boundary Element Methods that is used to model the exterior citeJSV,Brebbia,JOSAA leading to a system of the form \[
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix},
\] where \(A\) is Symmetric Positive Definite (SPD), banded with a reasonably wide band, and real valued while \(D\) is dense and complex valued. Traditionally, libraries that address this problem deal with the different parts explicitly as part of the solver.

Our approach is fundamentally different. This system can be viewed as a \(2 \times 2\) matrix of matrices. Each of the submatrices has a different numerical property, a different datatype, and/or further structure. The descriptors for nodes in the Hyper-Matrix can include fields that describe such special structural and/or numerical properties.

**Optimization of scheduling for SMPs and multi-core architectures** The advent of multi-core systems means that everyone will need to start worrying about parallelism. Traditional libraries will need to be recoded to improve opportunities for parallelism. Consider the following slides from a recent talk by Jack Dongarra titled “The Challenges of Multicore and Specialized Accelerators” [16]:
The slide on the left acknowledges that major changes to LAPACK are in order. The slide on the right shows an outline of what must be included in implementations of a Cholesky factorization that includes pipelining of tasks only in the column direction. (2D work partitioning and pipelining will be required on processors with many cores, much like it is on distributed memory architectures, further complicating matters) Those who know how LAPACK and ScaLAPACK are coded can imagine the complexity that these new features will introduce if dense linear algebra libraries continue to be coded in the tradition of LINPACK, LAPACK, and ScaLAPACK.

Our approach is fundamentally different. By expressing matrices as Hyper-Matrices and writing code to obey the hierarchy, the code becomes simpler than the original LAPACK code. Furthermore, the identification of blocks as fundamental units of data creates the opportunity to present operations on the matrix as a collection of operations on blocks. Dependencies between those can then be analyzed on a block-by-block bases, allowing techniques used by super-scalar architectures [1] for scheduling computation on scalars to be used to schedule these units of computation on a block basis. This will allow optimization of the scheduling as well as tracking of dependencies to be incorporated transparent to the library developer and user.

### 4 Defining and Manipulating Hyper-Matrices

In this section, we define Hyper-Matrices and discuss an API for creating and manipulating Hyper-Matrices in the C programming language, summarizing the contents of a technical report on the subject [40].

#### 4.1 Notation

We will use the common convention that matrices are denoted by capital letters, vectors by lower case letters, and scalars by Greek letters. Given a matrix $X$, the functions $m(X)$ and $n(X)$ return the row and column dimensions of $X$, respectively.

Consider the operation $C = AB + C$, where $m(C) = m(A)$, $n(C) = n(B)$, and $n(A) = m(B)$. The elements of $C$, $\{c_{ij}\}$, are updated with the elements of $A$, $\{\alpha_{ip}\}$, and $B$, $\{\beta_{pj}\}$ by the algorithm in Fig. 1(a).

It is common in the description of high-performance algorithms to partition matrices into blocks of matrices:

$$C = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1N} \\ C_{21} & C_{22} & \cdots & C_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ C_{M1} & C_{M2} & \cdots & C_{MN} \end{pmatrix}, A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1K} \\ A_{21} & A_{22} & \cdots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{MK} & A_{M2} & \cdots & A_{MK} \end{pmatrix}, \text{ and } B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1N} \\ B_{21} & B_{22} & \cdots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{NK} & B_{K2} & \cdots & B_{KN} \end{pmatrix},$$

where $m(C_{ij}) = m(A_{ip})$, $n(C_{ij}) = n(B_{pj})$, $n(A_{ip}) = m(B_{pj})$. Then matrix-matrix multiplication, in terms of these blocks, can be implemented by the algorithm in Fig. 1(b). Multiple layers of blocking can be imposed upon matrices by further partitioning the blocks $C_{ij}$, $A_{ip}$, and $B_{pj}$ into finer blocks.

#### 4.2 Architectures with multi-level memory

With the advent of cache-based processors, a major concern became how to amortize the cost of moving data between memory layers. Let us consider a typical multi-level memory as depicted in Fig. 2. Inherently, data must be moved...
for $i = 1 : m(C)$
    for $j = 1 : n(C)$
        for $p = 1 : n(A)$
            $\gamma_{ij} = \gamma_{ij} + \alpha_{ip}\beta_{pj}$
        endfor
    endfor
endfor

for $i = 1 : M$
    for $j = 1 : N$
        for $p = 1 : K$
            $C_{ij} = C_{ij} + A_{ip}B_{pj}$
        endfor
    endfor
endfor

for $p = 1, \ldots, K$
    for $i = 1, \ldots, M$
        $T = A_{ip}^T$
        for $j = 1, \ldots, N$
            $C_{ij} = T^TB_{pj} + C_{ij}$
        endfor
    endfor
endfor

Figure 1: Algorithms for computing $C := AB + C$.

Figure 2: The hierarchical memories viewed as a pyramid.

from memory layers lower in this pyramid into, ultimately, the registers in order for the CPU to perform floating point operations on those data. Since such movement represents unproductive overhead, it becomes important to reuse data that is up the memory pyramid many times before allowing it to slide back down, while simultaneously hiding cost by prefetching data.

One of the most ideal operations that allows such amortization of data movement is the matrix-matrix multiplication. In the most simple terms, this operation performs $O(n^3)$ operations on $O(n^2)$ data. Since this operation can be decomposed into smaller blocks that fit a given target memory layer, there is the clear opportunity of amortizing the cost of the data movement over many operations. It has been shown that most dense linear algebra operations can be cast in terms of matrix-matrix multiplication, which then allows portable high performance to be achieved [17, 5].

### 4.3 High-performance implementation of matrix-matrix multiplication

Let us briefly review the anatomy of a typical high-performance implementation of matrix-matrix multiplication.

The fastest implementations [2, 23, 28, 47, 11] of matrix-matrix multiplication $C := AB + C$ use algorithms similar to those in Fig. 1(c). We recognize this as a simple permutation of the loops. Details of why the loops are structured in this specific order go beyond the scope of this proposal. Also, in order to reduce TLB misses and other unwanted cache behavior, $A_{ip}$ is typically transposed and copied into a contiguous buffer, $T$. This reduces the number of TLB entries required to address this block and allows inner-products of columns of $T$ and $B_{pj}$ to be used, which access memory mostly contiguously, for the computation of elements of $C_{ij}$. The block size of $A_{ip}$ is chosen so that one such submatrix fills most of the L2 cache (unless the amount memory addressable by the TLB is less that the size of the L2 cache, in which case this becomes the limiting factor)\(^1\). Since often $C = A^TB + C$, $C = AB^T + C$, and/or $C = A^TB^T + C$ are also encountered in applications, locks of $B$ and/or $C$ may also need to be packed and/or transposed to improve data locality.

The typical observation now is that if matrices are stored by blocks, then this packing is no longer required. Moreover, within this storage scheme, transposition can be made cheaper, for example by transposing in-place. A

\(^1\)In some implementations, $A_{ip}$ is chosen to fill most of the L1 cache instead.
second observation is that with the introduction of more and more memory layers, multiple levels of blocking may become required in order to optimally implement matrix-matrix multiplication [28].

4.4 An API for storing and manipulating Hyper-Matrices

In this section we propose a simple API for the C programming language, merely to illustrate the issues. The approach is language independent.

Many papers describe the concept of hierarchically stored matrices [19, 37, 38, 13, 36, 34, 3, 45, 49, 22]. What is typically absent from these papers is any example of how to instantiate such data structures in code and how to code operations over such data structures (although algorithms are frequently given). One exception is those efforts that utilize the ability of C++ to hide indexing details [45]. Even there, much complexity is required under the covers in order to index into the data structures. In this section, we show how an API that mirrors the way hierarchical matrices are naturally explained allows many of these indexing intricacies to disappear. We do so by discussing a simple extension to our FLAME API for the C programming language, FLAME/C, as a concrete example of an instantiation of this idea [26, 9, 10].

4.5 Describing matrices in FLAME/C

The FLAME/C API hides details like storage method and matrix dimensions from the user by utilizing object based programming techniques much like the Message-Passing Interface (MPI) and other software packages [24, 44, 6, 46]. These details are stored in a descriptor and hidden from the programmer. They can be accessed through inquiry routines. In its current implementation, it is assumed that all matrices are stored using column-major storage as prescribed by Fortran.

FLAME/C provides the following call to create a descriptor and storage for an \( m \times n \) matrix of type \( \text{double} \), where integers \( m \) and \( n \) indicate the matrix row and column dimensions, respectively:

\[
\text{FLA}_\text{Obj} \text{ C};
\]

\[
\text{FLA}_\text{Obj}_\text{create}( \text{FLA}_\text{DOUBLE}, m, n, \&\text{C} );
\]

Given that now \( \text{C} \) is a descriptor for a FLAME object (declared to be of type \( \text{FLA}_\text{Obj} \)), the following calls then extract the various attributes:

\[
\text{datatype}_\text{C} = \text{FLA}_\text{Obj}_\text{datatype}( \text{C} );
\]

\[
\text{m}_\text{C} = \text{FLA}_\text{Obj}_\text{length}( \text{C} );
\]

\[
\text{n}_\text{C} = \text{FLA}_\text{Obj}_\text{width}( \text{C} );
\]

\[
\text{ldim}_\text{C} = \text{FLA}_\text{Obj}_\text{ldim}( \text{C} );
\]

\[
\text{buf}_\text{C} = ( \text{double*} ) \text{FLA}_\text{Obj}_\text{buffer}( \text{C} );
\]

Notice that the last call will set \( \text{buf}_\text{C} \) to the address of where the elements of the matrix are stored, so that the \((i, j)\) element can be set by the assignment

\[
\text{buf}_\text{C}[ j*\text{ldim}_\text{C} + i ] = < \text{value of } (i,j) \text{ element} >
\]

4.6 Supporting Hyper-Matrices with FLASH

The logical extension to the above API call that will allow the support of Hyper-Matrices is to allow each element of a matrix created by \( \text{FLA}_\text{Obj}_\text{create} \) to itself be a matrix. Thus the call

\[
\text{FLA}_\text{Obj}_\text{create}( \text{FLA}_\text{MATRIX}, m, n, \&\text{C} );
\]

creates an \( m \times n \) matrix where each element of \( \text{C} \) is a descriptor for a (not yet created) matrix \( C_{ij} \). We will refer to the FLAME/C API augmented with this as FLASH: Formal Linear Algebra Scalable Hierarchical API.

Given arrays \( m_\text{s} \) and \( n_\text{s} \) where each element, \( C_{ij} \), is to be of size \( m_\text{s}[i] \) by \( n_\text{s}[j] \), the following loop then creates the descriptors for the elements.
ldim_C = FLA_Obj_ldim(C);
buf_C = {FLA_Obj*} FLA_Obj_buffer(C);

for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    /* Create a matrix C_ij of size m[i] x n[j] */
    FLA_Obj_create(FLA_DOUBLE, ms[i], ns[j], &buf_C[j*lda_C + i]);

Naturally, by taking some or all submatrices $C_{ij}$ of type FLA_MATRIX, further layers of a hierarchical matrix can be created.

It is beneficial to have one subroutine call that creates all the layers of the hierarchy. More importantly, having a single subroutine call to create the hierarchy allows us to ensure that the hierarchy created will possess properties related to conformity. (As part of the proposed project these properties will be thoroughly studied, hopefully leading to necessary and sufficient conditions.) One example of such a hierarchy is created by the call

    FLASH_Obj_create(int datatype, int order, int m, int n, int depth,
                     int *Blksizes, FLA_Obj &C);

Here, the datatype of the elements in the leaf matrices is given by datatype. The order in which blocks are ordered is given by order. The flat matrix to be created is to be of size $m \times n$, where $m$ and $n$ are passed as arguments $m$ and $n$, respectively. The argument depth equals the depth of the hierarchical matrix to be created. Blksizes is an array of integers that indicates the size of the matrix at each level.

A more intuitive way of thinking about creating the hierarchy may be to think of partitioning the matrix with increasingly finer sieves while maintaining the constraint that the size of the current sieve must be a multiple of the size of the next smaller sieve. Figure 3 is a diagram of how the hierarchy is created with sieves. That figure illustrates a hierarchical matrix created with the call

    FLASH_Obj_create( datatype, FLASH_GENERALIZED_MORTON, 18, 23, 2, Blksizes, &C )

where int Blksizes[2] = {2, 5};. In this particular example, the top matrix is a $[18/(2 \times 5)] \times [23/(2 \times 5)] = 2 \times 3$ matrix of matrices. Each of those matrices is a $2 \times 2$ matrix, except the last column. At the leaves are matrices of dimension (at most) $5 \times 5$.

In our above explanation we do not “pad” the matrix to become of a dimension that is a multiple of the largest sieve. However, the interface doesn’t necessarily prescribe the implementation. Additional memory can always be used to pad the matrices if there is some advantage to doing so.

### 4.7 Storage schemes

Having described how to build the hierarchy, we, next, need to consider how to store the elements of the matrix. Here what really matters is the order in which the data in the leaf matrices is stored and the order in which these blocks are
Figure 4: Generalized Morton order: The lowest level blocks are $5 \times 5$ matrices stored in column-major order. The next level consists of $3 \times 3$ blocked matrices, stored in column-major order. Finally, those blocks are stored in column-major order themselves. Notice that the traditional Morton order can be achieved by picking the blocking at each level to be $2 \times 2$ for $\log_2(n)$ levels, storing each level in row-major order.

stored relative to each other. Street wisdom is that the better the locality of the data at the various levels, the better different layers of the memory hierarchy can be used.

**Recursive Block Storage - Morton(Z) Ordering** A Morton (Z) ordering [13, 22, 45] (with blocks at the leaves) can be achieved as follows: Store the $b \times b$ leaf matrices in some format, where the $b^2$ elements are stored in contiguous memory. Form $2 \times 2$ matrices of such blocks at the next level, storing these blocks in contiguous memory ordering them in row-major order. Continue like this, building $2 \times 2$ matrices at each level.

The approach illustrated in Fig. 4 is a variant of this. Given an $n \times n$ matrix, it can be described as follows: An $n \times n$ item buffer of continuous memory is allocated. Given a set of blocking sizes \( \{b_0, b_1, \ldots, b_{L-1}\} \), the flat matrix is partitioned into $B \times B$ blocks where $B = \prod_{i=1}^{L-1} b_i$ blocks. Assuming for simplicity that $n$ is an integer multiple of $B$, these top-level blocks are assigned $B \times B$ items of contiguous memory, in a column-major order. Each such $B \times B$ block is now further subdivided using blocking sizes \( \{b_1, b_2, \ldots, b_{L-1}\} \). Leaf matrices, of size $b_{L-1} \times b_{L-1}$, are stored in column-major order. It is not hard to see that this is modification of the Morton (Z) ordering, in that it uses column-major ordering and at each level the blocked matrices need not to be $2 \times 2$.

**Canonical Storage** The following calls allows us to attach a conventional column major ordered matrix to an existing hierarchy. This creates a hierarchy in which the leaf matrices act as views into different parts of the original matrices.

```c
FLASH_Obj_create_without_buffer( datatype, m, n, depth, *Blksizes, &C );
FLASH_Obj_attach_buffer( *buf, ldim, &C );
```

More precisely, the flat matrix is stored in column major order. At each level, the submatrices are simply submatrices of this matrix. The net effect is that the leaf matrices are stored in column-major order, with a leading dimension that equals the leading dimension of the flat matrix.

*The point is that, without requiring any recoding of algorithms, a multitude of different physical storage schemes can be explored and exploited.*

### 4.8 Filling the matrix

An easy argument for *not* accepting alternative data structures is to point out that filling such matrices with data is simply too complex for applications. This would appear to favor the use of C++ since it would allow indexing to be hidden from the user. We propose an alternative solution that is based on submitted and extracting submatrices to and from a Hyper-Matrix so that the application does not need to be aware of the intricacies of the datastructure. For details, see [18].
void FLASH_Chol( FLA_Obj A )
{
    FLA_Obj ATL, ATR, A00, A01, A02,
        ABL, ABR, A10, A11, A12,
            A20, A21, A22;
    if ( FLA_Obj_datatype( A ) != FLA_MATRIX )
        return FLASH_Chol( FLA_LOWER_TRIANGULAR, A );
    FLA_Part_2x2( A, &ATL, &ATR,
        &ABL, &ABR, 0, 0, FLA_TL );
    while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) )
    {
        FLA_Repart_2x2_to_3x3( ATL, &ATR, &A00, &A01, &A02,
            &A10, &A11, &A12,
                &ABL, &ABR, &A20, &A21, &A22, 1, 1, FLA_BR );
        FLASH_Chol( FLASH_MATRIX_AT( A11 ) );
        FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
            1, 0, A11, A21 );
        FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_N_TRANSPOSE, FLA_MINUS_ONE, A21, 1, A22 );
    }
}

Figure 5: Cholesky factorization of a matrix that is stored as a Hyper-Matrix using the FLASH API.

Our approach is to enable a user to enter data in a hierarchical matrix using routines that hide the storage from the user. We do not advocate conversion of storage nor of transparently allowing users to access element \((i, j)\). The purpose of the proposed project is to see how libraries and interfaces are changed when the hierarchical storage is embraced rather than hidden.

4.9 FLAME algorithms for Hyper-Matrices

For an example of how simple code becomes when matrices are stored hierarchically using the proposed API, we give a right-looking Cholesky factorization, in Fig. 5. (This example is representative of many BLAS and LAPACK level operations.) To understand the code, it suffices to recognize that the algorithm sweeps through the top-level matrix from the top-left to the bottom right. If the top-level matrix is a leaf matrix, a standard Cholesky factorization is called. If not, the diagonal element (which is a matrix itself) is factored recursively and BLAS-like calls are used to update the submatrices \(A_{21}\) and \(A_{22}\). All intricate indexing is hidden.

5 Research Focus Areas and Proposed Work

In this section, we discuss a few of the primary research topics that will be pursued as part of the proposed project.

5.1 Implementation of FLASH API

Status: A prototype implementation has been implemented. This implementation merely manages blocks. If does not have the ability to tag submatrices with properties and to manage parallelism. It incorporates only a few linear algebra operations. It does not yet interface to low-level kernels coded by Kazushige Goto.
Figure 6: Preliminary investigations on a 12 CPU SMP Itanium2 server (1.5GHz, 6GFLOPS/sec peak per CPU.)

Left: Comparison of serial GotoBLAS R1.6 DGEMM routine and FLAME matrix-matrix multiplication implemented by calling GotoBLAS low-level kernels to compute smaller matrix-matrix multiplications. This experiment shows that performance is essentially the same (minor variation comes from slightly different choice of strategy.) Also shown is a matrix-matrix multiplication that is implemented as a triple loop over matrices that are blocked into 128 × 128 submatrices. For each smaller multiplication with one submatrix from each of A, B, and C the GotoBLAS DGEMM is called. This means that submatrices of A and B are “repacked” multiple times. The overhead for this is reflected in the degradation of performance. It is this degradation that will be eliminated by calling low level kernels and reusing packed submatrices, as part of the proposed project.

Right: Parallel implementation of Cholesky factorization. A first set of three experiments links the implementations to Intel’s MKL (8.1) library. Here, Multithreaded MKL refers to a call to the MKL routine dpotrf (Cholesky factorization), letting all parallelism come from the multithreaded MKL library; blk + Multithreaded MKL BLAS indicates a sequential right-looking blocked Cholesky factorization algorithm that calls multithreaded BLAS from the MKL library; and data-flow 2D + serial MKL refers to a algorithm that performs a Cholesky factorization by blocks, implemented with FLASH but without the benefits of FLASH, statically scheduled so that the threats form a two-dimensional mesh, linked to sequential MKL BLAS. The second set of two experiments compares a sequential right-looking blocked Cholesky factorization implemented with FLAME and linked to multi-threaded GotoBLAS (1.06) to the same statically scheduled parallel algorithm now linked to the GotoBLAS. The first set shows that when linked to a poor implementation of the BLAS, the multi-threading by blocks shows merit. The second set shows that the overhead of calling DGEMM for each set of blocks instead of low-level kernels carries an overhead. As part of the proposed project, such algorithms by blocks will be recoded using FLASH, allowing (1) scheduling to happen transparently; (2) packed blocks to be reused; and (3) low-level kernels to be called directly. It is our expectation that this will make FLASH-based multi-threaded implementations highly competitive.

Proposed work: Full-blown implementation of infrastructure to the point where the below-mentioned tasks can be pursued.

5.2 Low-level kernel development

Status: Low-level kernels that are building blocks for the fastest BLAS libraries available under Open Source license, the GotoBLAS, have been developed. Prototype interfaces to these kernels via the FLAME API have been developed. The performance of FLAME interfaced to these low-level kernels is assessed in Fig. 6(left). (See caption for interpretation of the data.)

Proposed work: A number of developments will be pursued:
Proposal of an informal standardization of interfaces to low-level kernels. This would allow other research, by projects at other institutions, to benefit from access to the performance of low-level kernels without the overhead of calls to standard BLAS routines.

Development of kernels specifically for the setting when matrices are stored and used by blocks. For example, current BLAS implementations keep blocks of \( \mathbf{A} \) in the L2 cache since this amortizes the cost of packing these blocks from a standard column-major order. If matrices are already stored by blocks, it is conceivable that keeping such (smaller) blocks in the L1 cache will achieve better performance. Also, when storing matrices by blocks it becomes convenient to insist on square blocks (for reasons of conformal dimensions). This may change the design of the low-level kernels. A complete evaluate of the issues will be pursued.

A critical issue that affect performance is the ability to create work space that is not just logically contiguous, but also physically contiguous. Is it the case that storage by, and computation with, blocks simplifies this issue, since obtaining small physically contiguous workspace is easier than the large buffers that are currently needed?

Is there a benefit to “hanging” already packed copies of blocks off the object that describes a submatrix in the hierarchy?

5.3 Evaluation of the benefits of storage by blocks for sequential, SMP, and multi-core architectures, with complex multi-level memories.

**Status:** A prototype implementation of the FLASH API has been built upon FLAME [40]. An interface from FLAME/C to low-level kernels for matrix-matrix multiplication (copy and multiplication with small subproblems) has been created. An undergraduate, Bryan Marker, has shown that on sequential and small SMP systems this interface allow performance to be achieved by matrix-matrix multiplication that essentially matches that of the GotoBLAS, which are implemented at a much lower level of abstraction (see Fig. 6(left)). For these experiments, matrices are not stored by blocks nor are they manipulated as hierarchical matrices. In addition, Ms. Merche Marques, a Ph.D. student in Spain, has been using a prototype FLASH interface to code out-of-core dense linear algebra operations.

**Proposed work:** A thorough evaluation of the benefits of storing and/or viewing the matrices hierarchically, of the benefits (or lack thereof) of storage via Morton ordering or generalized Morton orderings, targeting multi-level memory architectures including multiple levels of cache, shared memory, and disk (out-of-core computation). The evaluation will also span matrices with special structure and how the approach reduces memory usage.

5.4 Hyper-Matrices for Sparse Direct Methods

**Status:** In a somewhat related NSF sponsored project, mentioned in Section 12, we are studying the use of a datastructure we call a Unassembled Hyper-Matrix (UHM). The idea is that if the datastructure that stores a matrix reflects the hierarchy of refinement in a dynamic FEM application, recomputation of a sparse Cholesky or LU factorization can be limited to parts of the hierarchy that changes with a refinement. Key is the insight that the matrices should be stored in an element-by-element fashion without assembling them explicitly.

**Proposed work:** As part of the proposed project we intend to use Hyper-Matrices coded with FLASH to assemble an UHM into a global matrix. This will allow us to compare and contrast methods that are developed as part of the UHM project, traditional methods for sparse direct solution, and new methods that mimic the traditional methods but organize the computation more naturally by obeying the hierarchy in the matrix that comes from the successive substructuring.

5.5 Transparent management of dependencies and data between tasks on SMP and multi-core architectures

**Status:** Partitioning a matrix into square blocks and scheduling computation over those blocks provides for a convenient and effective way of creating tasks. The challenge lies with (1) keeping the programs simple; (2) scheduling of the resulting blocks of computation; (3) resolving dependencies; (4) overcoming overhead that is incurred by repeating
The details of how to schedule operations will be a topic of study as part of the proposed work. For example, if many operations update the same block (e.g., by calls to FLASH_Syrk from different iterations), it may become beneficial to privatize a copy of the target block, and adding the block eventually in an atomic operation. Again, similar techniques used on super-scalar architectures will be used for inspiration. For example, the notion of recognizing “macro-operations” composed of multiple independent tasks, similar to techniques used on VLIW architectures [20], can be incorporated. (Recognize that all subcomputations performed in the calls to FLASH_Trsm and FLASH_Syrk in Fig. 5 form a macro-operation. This demonstrates how, again, techniques for super-scalar architectures can be easily and efficiently implemented in software, on a block-by-block basis.) Finally, these types of techniques can be used to achieve processor affinity, binding computation to a specific processor or core to improve performance.

We expect that storage by blocks will also benefit the programming of multi-core architectures with many cores (64-128 cores per chip). It is likely that such architectures will include memory local to each core. The blocks become units of communication between cores, possibly simplifying the communication mechanism. For example, a block and information associated with it could be restricted to one page so that communication of pages becomes the basic unit. A simpler communication mechanism may allow communication to be more efficient.

Proposed work: Many different approaches to this challenging problem will be pursued.

One approach centers on the Tomasulo algorithm, a simple but highly effective approach for scheduling operations on super-scalar architectures. Let us illustrate the approach on a $3 \times 3$ matrix of scalars, $A$:

$$
\begin{pmatrix}
\alpha_{00} & \ast & \ast \\
\alpha_{10} & \alpha_{11} & \ast \\
\alpha_{20} & \alpha_{21} & \alpha_{22}
\end{pmatrix}.
$$

If the algorithm in Fig. 5 is applied to this scalar matrix, the operations in Fig. 7 will need to be performed. That figure also hints at how the algorithm keeps track of dependencies and schedules the operations.

The simple observation is that this table can be built on a block-by-block basis when matrices are viewed and/or stored by blocks. This can occur in software, since the overhead of maintaining the table is amortized over a large amount of computation when blocks are reasonably large. It can occur transparently, since placing operations on the table can be achieved inside of calls to routines like FLASH_Chol, FLASH_Trsm, and FLASH_Syrk and blocks can be identified by their descriptors.

The details of how to schedule operations will be a topic of study as part of the proposed work. For example, if many operations update the same block (e.g., by calls to FLASH_Syrk from different iterations), it may become beneficial to privatize a copy of the target block, and adding the block eventually in an atomic operation. Again, similar techniques used on super-scalar architectures will be used for inspiration. For example, the notion of recognizing “macro-operations” composed of multiple independent tasks, similar to techniques used on VLIW architectures [20], can be incorporated. (Recognize that all subcomputations performed in the calls to FLASH_Trsm and FLASH_Syrk in Fig. 5 form a macro-operation. This demonstrates how, again, techniques for super-scalar architectures can be easily and efficiently implemented in software, on a block-by-block basis.) Finally, these types of techniques can be used to achieve processor affinity, binding computation to a specific processor or core to improve performance.

We expect that storage by blocks will also benefit the programming of multi-core architectures with many cores (64-128 cores per chip). It is likely that such architectures will include memory local to each core. The blocks become units of communication between cores, possibly simplifying the communication mechanism. For example, a block and information associated with it could be restricted to one page so that communication of pages becomes the basic unit. A simpler communication mechanism may allow communication to be more efficient.

Figure 7: An illustration of the Tomasulo algorithm for the Cholesky factorization of a $3 \times 3$ matrix of scalars. A table is built of all operations and their input and output variables. A $\sqrt{\ast}$ indicates the value is available. When all parameters are $\sqrt{\ast}$ed, the operation is scheduled for execution. Upon completing the operation, the output variable is checked everywhere in the table until after it appears again as an output variable. Naturally, the scheduling of operations that are ready to be performed affects the exact order in which subsequent operations are identified as ready. In particular, one would expect some level of pipelining to occur.  

behind-the-scenes data rearrangements; and (5) reducing the movement of blocks of data between cores and/or CPUs of an SMP (“binding” computations and data to specific cores). Initial experiments, performed by our collaborator Prof. Gregorio Quintana at the Univ. of Jaume I (Spain), are reported in Fig. 6(right).
The need for simple and transparent, but powerful, scheduling and optimization solutions becomes particularly obvious when one realizes that, much like for distributed memory architectures, 2D distribution of work and data to processors (cores) becomes necessary. Add to this the required hand-management of data movements between local memories of cores... (Space limitations keeps us from presenting the data that supports this insight.)

We intend to model different scenario in an effort to analyze future architectural designs.

5.6 Impact on productivity/library development

Status: Coding algorithms for hierarchical matrices is complex when one codes at a low level of abstraction. An indication of how difficult can be found in the recent papers by E. Elmroth et al [19, 38]. These papers review the benefits of recursive algorithms and discusses storage by blocks. While it discusses the implementation of many recursive algorithms, it only hints at the implementation of a few such algorithms when matrices are stored hierarchically by blocks. One recursive algorithm discussed is the solution of the triangular Sylvester equation $UX + XR = C$, where $U$ and $R$ are (square) upper triangular matrices and $X$ and $C$ are rectangular matrices. Notice that if $U$ is $m \times m$ and $R$ is $n \times n$, then $C$ and $X$ are both $m \times n$. In this equation, $U$, $R$, and $C$ are inputs, and $X$ is an output. It is customary that the solution $X$ overwrites input matrix $C$. No implementation is given in that paper for the case where matrices are stored hierarchically by blocks. We take this to indicate that this algorithm was too complex to implement easily by the authors, who were experienced in implementing algorithms for matrices stored hierarchically by blocks.

In [42], we show how a family of algorithms can be derived for this same operation using the FLAME approach to deriving algorithms. Also in that paper, we show how the FLAME/C API can be used to implement them. The best algorithms are competitive with those discussed in [19, 38], when matrices are stored in column-major order.

To gain initial insight into how the FLASH API can facilitate the implementation of these kinds of algorithms, we coded both the recursive algorithm in [19] and the best algorithm (which combines iteration and recursion) in [42] (Algorithm C3 in that paper). In a matter of less than an hour, both these algorithms were implemented and tested for correctness. We use this as an indication that the API facilitates rapid prototyping of recursive algorithms for matrices that are stored as hierarchical matrices. This means that most of the effort in the proposed project can be focused on pursuing intellectually interesting problems and the creation of reasonably complete prototype libraries rather than expending much energy on tedious coding.

Proposed work: The only way that storage hierarchically by blocks will gain acceptance in the scientific computing community is if (1) it is demonstrated that there is a performance benefit; (2) simple interfaces are developed that shield the application programmer from the details of the datastructure; and (3) a reasonably complete library is made available. The only way that reasonably complete libraries will become available through contributions of mathematicians that develop numerical libraries is if the coding effort is shown to be manageable. We intend to achieve these goals by developing a library with functionality similar to the BLAS and LAPACK that stores matrices as Hyper-Matrices as an extension of the FLAME library effort. Target architectures will include current and future sequential, SMP, and multi-core architectures.

6 The Team

Dr. Robert A. van de Geijn (PI), Professor of Computer Sciences and Member of the Institute for Computational and Applied Mathematics (ICES). He has almost two decades of experience with high-performance and parallel computing.

Mr. Kazushige Goto (CoPI), Research Scientist, Texas Advanced Computing Center (TACC). Mr. Goto received his Masters in Power Engineering from Waseda University (Japan) in 1994. He is the developer of the fastest BLAS libraries, the GotoBLAS, for essentially all current microprocessor architectures. He will develop specialized “inner-kernels” tailored to hierarchically stored matrices.

Field Van Zee, Research Scientist Associate II, Dept. of Computer Sciences, UT-Austin. Mr. Van Zee is the primary library developer for the FLAME project and will be in charge of fully integrating the FLASH library with the FLAME library, which is available under LGPL license.
Dr. Enrique Quintana-Ortí, Associate Professor of Computer Architecture, University of Jaume I, Spain. Dr. Enrique Quintana has published extensively in the areas of high-performance computing, parallel computing, and algorithms for control theory (many of which are structured sparse). As part of the project, Dr. Enrique Quintana will visit UT-Austin for two weeks every year funded by the grant.

Dr. Gregorio Quintana-Ortí, Associate Professor of Computer Science, University of Jaume I, Spain. Dr. Gregorio Quintana is a frequent collaborator on the FLAME project and contributed preliminary results to the present proposal. As part of the project, Dr. Gregorio Quintana will visit UT-Austin for two weeks every year funded by the grant.

Ms. Mercedes Marques, Ph.D. student University of Jaume I, Spain. Ms. Marques will continue to pursue the development of out-of-core dense linear algebra libraries for control theory based on the FLASH API. She is funded through other means. She will be encouraged to visit UT-Austin to collaborate as part of the proposed project.

**Student involvement** The project will involve graduate and undergraduate students. Fig. 6(left) was contributed by Mr. Bryan Marker who is pursuing an undergraduate honors project related to the proposed project. Mr. Ernie Chan, a second year graduate student, is pursuing the research related to multi-core architectures possibly towards his dissertation topic.

**Involvement from other members of the FLAME team** is expected as these members show an interest.

7 Project Management
The team members will work closely together on all aspects of the proposed project. Weekly seminars will be used to track the progress of the participants.

While the proposed project primarily concentrates on making a contribution to computer science, there is a great potential for impact on industry. To ensure this impact is realized, a number of distinguished researchers from industry have agreed to serve on an Industrial Advisory Committee for the Formal Linear Algebra Methods Environment (FLAME) project, which is the umbrella project that will include the proposed work. Many of these researchers are already actively involved with the team at UT in research related to the practical implications of the proposed project. An annual workshop to discuss research results with this advisory team will be organized.

**Industrial Advisory Team:** The FLAME project, of which the proposed project will be a part, includes significant collaboration with industrial researchers that builds on established collaborative research [27, 26, 8, 28, 30, 29]. These researchers include Dr. Andrew Chapman (NEC Solutions (America), Inc.), Dr. John Gunnels (IBM), Mr. Mike Kistler (IBM), Dr. Greg Henry (Intel), Dr. Tim Mattson (Intel), Dr. Daniel S. Katz (LSU and JPL), Mr. Jim Nagle (National Instruments), and Dr. Rob Schreiber (HP Labs).

8 Impact
The projects related to FLAME are having a broad impact on research in linear algebra libraries and computational science:

**Impact on Linear Algebra Libraries:** The approach demonstrates and enables a deeper level of understanding of the algorithmic, analysis, and software engineering issues related to linear algebra libraries.

**Impact on Applications:** The resulting libraries, available under Open Source license, and tools benefit the high-performance computing community.

**Impact on Education:** The simple interfaces and abstractions will enable advanced topics in the field to be made accessible to new graduate students and even undergraduate students.

9 Dissemination of Results
The results of the proposed project will be disseminated through the usual means: refereed journal and conference papers, a working note series, an attractive project website and code available under Open Source. In addition, the research team will continue to meet frequently with collaborators from industry and will organize minisymposia on the developed techniques and tools. The work is expected to one PhD dissertation.
10 Related Work

The idea of using Hyper-Matrices for storing sparse matrices dates back to the early 1970s [43]. Unfortunately, at that time the now more conventional storage schemes were adopted instead. It was later revisited for distribution of matrices to distributed memory architectures in a dissertation by Tim Collins [14], supervised by J.C. Browne and R. van de Geijn.

Other related work has been mentioned throughout the proposal.

11 Proposer’s Relevant Research and Comparison with Long-term Goals

Proposer’s relevant research has been mentioned throughout the proposal. Here we show how the proposed projects builds upon and strengthens other on-going projects.

The FLAME project

The objective of the FLAME project is to transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike. Rather than being only a library, the project encompasses a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for mechanical derivation, implementation and analysis of algorithms and implementations.

Notation The key insight that enables the FLAME methodology is a new, more stylized notation for expressing loop-based linear algebra algorithms. This notation closely resembles how algorithms are naturally illustrated with pictures [26, 30].

Derivation The FLAME project promotes the systematic derivation of loop-based algorithms hand-in-hand with the proof of their correctness. Key is the ability to identify the loop-invariant: the state to be maintained before and after each loop iteration, which then prescribes the loop-guard, the initialization before the loop, how to progress through the operand(s), and the updates. To derive algorithms one fills out a “worksheet” [9].

APIs A number of APIs have been defined for representing the algorithms in different languages: LATEX, C (FLAME/C), C with MPI (PLAPACK2e), C with OpenMP (OpenFLAME), Matlab (FLAME@lab) [10]. Special needs specialization include fault-tolerance (FLARE) [31] and out-of-core computation (POOCLAPACK) [32, 33].

Analysis Assertion of correctness of algorithms and implementations in the presence of round-off error must include an assessment of numerical stability. In his dissertation, Paolo Bientinesi shows that the FLAME methodology can be extended to yield systematic stability analyses [7]. Systematic analysis of the cost of different algorithms and implementations has/is also being pursued by the FLAME project [25].

Tools A tool for mechanically deriving linear algebra algorithms, AutoFLAME, was developed as part of Paolo Bientinesi’s dissertation and is frequently used as part of the algorithm and library development efforts [7].

Libraries In addition to the contribution to science that the FLAME project represents, a broader impact comes from the libraries that have been released, including the libFLAME and GotoBLAS libraries.

12 Results from Prior NSF Support (Last 5 years)

12.1 Funding for FLAME

Since July 1, 2002, the FLAME project at UT-Austin has been funded by four NSF grants:


Robert van de Geijn (PI) (collaboration with Anthony Skjellum): Award ACI-0305163 for “ALGORITHMS: Collaborative Research: A Systematic Approach to the Derivation, Representation, Analysis, and Correctness of Dense

These projects are closely related to each other and the proposed project. Three Ph.D. students and one Masters student were supported in part by these projects:

Dr. Paolo Bientinesi recently completed his dissertation titled “Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.” His dissertation has been nominated for the ACM Best Dissertation Award.

Mr. Tze Meng Low advanced to candidacy in Dec. 2004. His dissertation will show how the techniques that underlie the FLAME project support a unified theory of loop transformations for the given problem domain.

Mr. Ernie Chan is a second year Ph.D. student and may pursue a topic related to the proposed project.

Mr. Field Van Zee completed a Masters degree in Spring 2006 and is now the chief software architect of the FLAME project.

Ms. Mercedes Marques, a Ph.D. student at the Univ. of Jaume I, Spain, is pursuing a dissertation topic related to these projects and is co-supervised by Prof. Enrique Quintana and Prof. Robert van de Geijn.

Mr. Bryan Marker, an undergraduate in the UTCS Turing Scholars honors program, is pursuing an undergraduate honors project related to the project.

These projects have already yielded a wealth of publications that acknowledge these grants: a complete draft of a book (publisher to be determined), six journal papers are in print (five to TOMS, one to SISC), one journal paper (to TOMS) that has been accepted and is being revised, and five journal papers (three to TOMS and two to Concurrency and Computation: Practice and Experience) are under review, in addition to a large number of conference papers. A beta release of the libFLAME library, with functionality that encompasses much of the BLAS and LAPACK, was announced Sept. 1, 2006. It is available under LGPL license. Materials developed as part of the project are regularly used in undergraduate and graduate classes offered at UT-Austin by the PI. For details, visit http://www.cs.utexas.edu/users/flame/.

12.2 Other funding


New project that investigates new techniques for storing and manipulating (directly) sparse matrices that arise from highly dynamic FEM applications.
References


