Programming for Locality and Parallelism with Hierarchically Tiled Arrays

Gheorghe Almasi(1), Luiz De Rose(1), Jose Moreira(1) and David Padua(2)

(1){gheorghe,laderose,jmoreira}@us.ibm.com
IBM T. J. Watson Research Center
P. 0. Box 218
Yorktown Heights, NY 10598-0218

(2)padua@uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

This paper introduces a new primitive data type, hierarchically tiled arrays (HTAs), which could be incorporated into conventional languages to facilitate parallel programing and programming for locality. It is argued that HTAs enable a natural representation for many algorithms with a high degree of locality. Also, the paper shows that, with HTAs, parallel computations and the associated communication operations can be expressed as array operations within single threaded programs. This, is then argued, facilitates reasoning about the resulting programs and stimulates the development of code that is highly readable and easy to modify. The new data type is illustrated using examples written in an extended version of MATLAB.

1.0 Introduction.

This paper introduces a new primitive data type which could be incorporated into conventional languages to facilitate parallel programing and programming for locality. This new data type facilitates the representation and manipulation of arrays that are organized as a hierarchy of tiles. These *hierarchically tiled arrays (HTAs)* are a generalization of the recursively blocked arrays arising in some linear algebra algorithms with a high degree of locality. Our proposal is to use HTAs to facilitate the expression of both locality and parallelism. In a nutshell, our idea is to distribute the outermost tiles of a hierarchically tiled array for parallelism, and used the inner tiles for locality and message aggregation. In the case of sequential programs all tile levels will be used for locality.

The two main sources of inspiration for this project were the extensive body of work on blocked linear algebra algorithms [Gust97, ABDD92] and two recently proposed languages, Co-Array Fortran [NuRe98] and Unified Parallel C (UPC) [CDCY99]. Our proposal follows these two languages in that it represents communication explicitly as array assignments. The use of array assignments to represent communication has at least two advantages over the library-based, approach of MPI [GrES99]. First, thanks to APL [Iver62] and Fortran 90 we have at our disposal a wealth of powerful array operators that can serve to unify and simplify the many communication and collective operations of MPI. Second, making the operations part of the language enables compiler support that simplifies the notation and improves error detection.

We, however, do not follow Co-Array Fortran and UPC in the use of the SPMD programming paradigm. Instead, our proposal resembles the programming model of the old SIMD machines, but instead of limiting the parallelism to simple arithmetic or logic array operations, we take advantage of the MIMD nature of today's parallel machines and allow in the expression of parallelism the use of complex array operations represented as user-defined functions. Abandoning the SPMD model has the drawback of removing some control on the parallelism from the programmer, but the single thread programming model has the great advantage of enforcing structure and leading to programs that are more readable and easier to develop and maintain. Furthermore, we expect that much of the potential loss of performance can be avoided with relatively simple compiler and run-time techniques.

Our approach differs from that of the High Performance Fortran [HiKT92, KoMe92] in that it makes all communication and array distribution is explicit and therefore it requires much less from the compiler than High Performance Fortran. Although making communication explicit complicates programming there is no better alternative at this time given the failure of High Performance Fortran. Furthermore, languages for parallel programming with explicit communication will always be necessary much in the same way that assembly language programming is still necessary today for conventional programming. The availability of a lower level language is useful as a fall back position whenever the compiler fails to do the right thing and as a means to experiment with alternative solutions that can later be incorporated into a compiler.

Hierarchically tiled arrays can be easily incorporated into several programming languages including Fortran 90, APL, and MATLAB. In this paper we focus on extending MATLAB with hierarchically tiled arrays for two main reasons. First is that an extended MATLAB system would make a great tool for prototyping parallel programs. Such a tool is sorely needed and although many MATLAB programmers may not be interested in parallelism, we believe that many parallel programmers would be interested in a good prototyping tool. The second reason is that MATLAB has many features that make it a convenient platform for a first implementation of our ideas.

In the rest of this paper, we describe hierarchically tiled arrays (Section 2), present mechanism for their representation in memory (Section 3) and then illustrate their use in programming for locality (Section 4) and parallelism (Section 5).

2.0 Hierarchically tiled arrays

In this section we define hierarchically tiled arrays (Section 2.1), and discuss how to build them (Section 2.2), access their components (Section 2.3), and how they can be used in expressions and values assigned to them (Section 2.4).

2.1 Definition of hierachically tiled array

We define a **tiled array** as *an array that is partitioned into subarrays in such a way that adjacent subarrays have the same size along the dimension of adjacency.* Although the literature usually assumes that array tiles have the same shape (Fig. 1(a)), we do not require this in our definition because there are important cases where using tiles of different sizes (Fig 1(b)) is advantageous. Notice that our definition implies that *m*-dimensional arrays are partitioned by (*m*-1)-dimensional hyper planes that are perpendicular to one of the dimensions. Furthermore, "randomly" partitioned arrays such as that shown in Fig. 2 do not fall under our definition of tiled arrays.

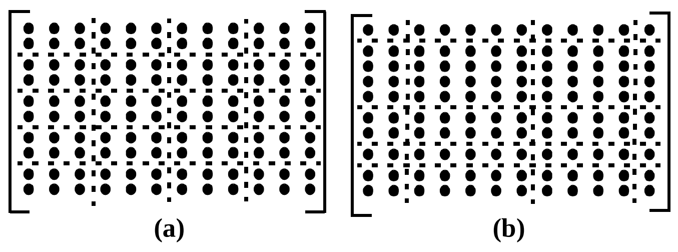

Figure 1. Two tiled arrays.

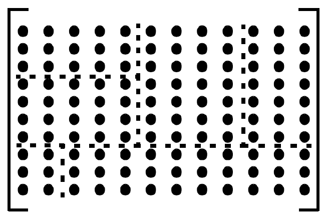

Figure 2. A partitioned array.

We define **hierarchically tiled arrays (HTA)** as *tiled arrays where each tile is either an unpartitioned array or a hierarchically tiled array.* Although this definition allows different tiles to be partitioned in different ways, most often HTAs will be *homogeneous*, that is adjacent submatrices at each level will not only have the same size as their neighbors along the dimension of adjacency, but they will also agree in the number and position of the partitions along that dimension.

A two-level hierarchy where neighboring tiles are partitioned differently, and therefore depicts a non-homogeneous HTA, is shown in Fig. 3(a). In this figure, the outer tiles are separated by the dashed lines and the inner tiles by the dotted lines. There are three mismatches in Fig. 3(a). One is between outermost tile {1,2}, which is not partitioned at all, and tile {1,1} which is partitioned into two parts along the vertical dimension which is the dimension of adjacency between these two tiles. The other two mismatches are between outermost tiles {1,1} and {2,1} and between tiles {2,1}, and {2,2}. Fig. 3(b) is an example of homogeneous HTA where the number of tiles and the sizes of all tiles match along the dimensions of adjacency.

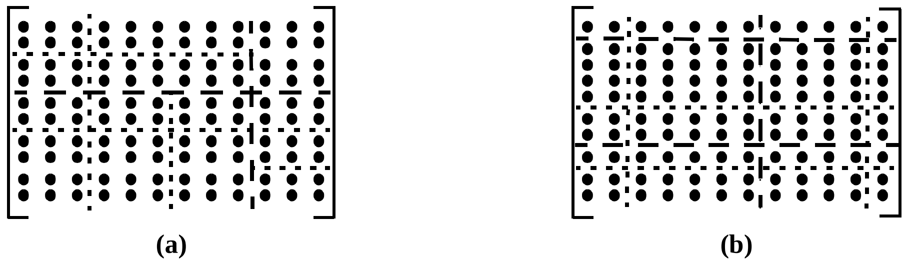

Figure 3. Two level tiled arrays.

2.2 Construction of HTAs

A simple way to obtain homogeneous HTAs is to tile the matrix at the lowest level of the hierarchy first and then proceed recursively by tiling the resulting array of tiles. This *bottom-up* process, illustrated in Fig 4, always generates homogeneous HTAs.

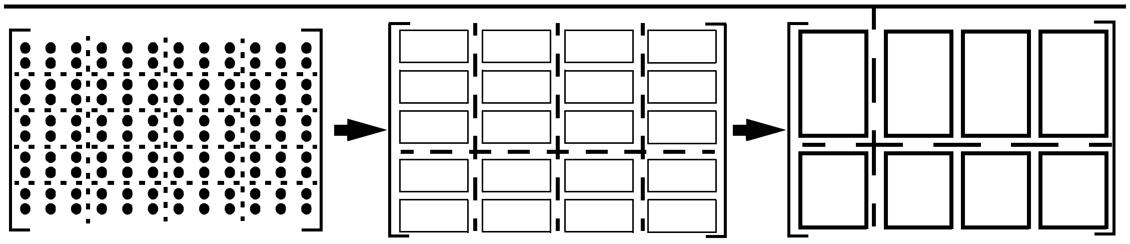

Figure 4. Bottom up tiling.

We can alternatively start from the top and successively refine each partition. The *top down* approach is more flexible than the bottom up approach in that it enables the generation of both homogeneous and nonhomogeneous HTAs.

In an interactive array language such as MATLAB, HTAs can be built following either approach if the appropriate functions are available. For the bottom up approach we define the function **`tile`** that accepts as parameters an *m*-dimensional HTA or unpartitioned array and m vectors, $\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_m}$, (one for each dimension of the HTA) and returns an HTA partitioned by the hyperplanes defined by $\boldsymbol{p_i}(k)$, $1 \le i \le m$, $1 \le k \le size(\boldsymbol{p}_i)$. These partition dimension *i* of the array right after element $\boldsymbol{p_i}(k)$. For example, given a 10×12 matrix, **`D`**, the statements

```
C = tile(D,[2,4,6,8],[3,6,9]);
B = tile(C,[3],[1,2,3]);                         (2.1)
A = tile(B,[1],[1]);
```

will generate the three HTAs shown in Fig. 4.

Our **`tile`** function is identical to MATLAB's **`mat2cell`** except that **`mat2cell`** can only be used to create one level of tiling.

For the top down approach we define the function **`hta`** which accepts *m* natural numbers as parameters, d_1, d_2, ... ,d_m, and returns a $d_1 \times d_2 \times ... \times d_m$ array whose elements are empty tiles that can hold HTAs or unpartitioned arrays. Before presenting an example of top down creation of HTAs, we need to describe how to address the tiles in an HTA. The outermost tiles of an HTA can be addressed using subscripts enclosed by curly brackets. An additional set of subscript should be added for each level of the HTA that needs to be addressed. Thus, the tile containing element **`E(5,4)`** if **`E`** is partitioned as shown in Fig. 1 (a) would be accessed as **`E{3,2}`**. Also, the inner tile containing element **`F(5,4)`** in an array **`F`** with the shape shown in Fig 3(b), would be addressed as **`F{2,1}{1,2}.`**

We can now illustrate the top down creation of HTAs. The top two levels of an array **`E`** with the shape shown in Fig. 3(a) could be created as follows:

```
G = hta(2,2);
G{1,1} = hta(2,2);                        (2.2)
G{2,1} = hta(2,3);
G{2,2} = hta(3);
```

and the elements of the upper left quadrant could be filled with two-dimensional arrays of normally distributed random number as follows:

```
G{1,1}{1,1} = randn(2,3);
G{1,1}{1,2} = randn(2,6);                 (2.3)
G{1,1}{2,1} = randn(2,3);
G{1,1}{2,2} = randn(2,6);
```

A drawback of the bottom up approach as illustrated in (2.1) is that it creates intermediate HTAs which are in most cases unnecessary. A reasonable compiler could have these temporary HTAs deleted after their only use in the creation sequence or could avoid their creation altogether by, for example, reversing the creation process into a top down form. As can be seen in the foregoing example, the top down approach does not suffer of this problem.

2.3 Addressing the scalar elements of an HTA.

We discuss next how to address the scalar elements of an HTA. The simplest way to address an element is to ignore all tiling and address the elements using conventional subscripting. For example, element 4,5 of an array, **`H`**, that has been tiled as shown in Fig. 1(b) can be addressed as **`H(4,5)`**. To use tiling for addressing a scalar element, we can use the curly bracket notation introduced above followed by conventional subscripts enclosed within parenthesis. The conventional subscripts specify the location of the element within the innermost tile in the hierarchy. Thus, element **`H(4,5)`** can also be addressed as **`H{2,2}(2,3)`**. We call *flattening* the mechanism that allows addressing an array ignoring the tile structure. Thus, we say that flattening enables the use of **`H(4,5)`** to access element 4,5 of array **`H`**. Flattening can also be applied at an intermediate level of the hierarchy. For example element 5,7 of an array **`A`** tiled a shown in Fig. 4 could be referenced as **`A(5,7)`**, or as **`A{1,2}{2}{3}(1,1)`** if all the levels of the tiling hierarchy are taken into account. We could also flatten the last level of the hierarchy and address the same element as **`A{1,2}{2}(5,1)`** or flatten the second level to get **`A{1,2}(5,4)`**.

2.4 Assignments and expressions involving HTAs.

The last topic to be discussed in this section is the meaning of assignment statements and expressions involving HTAs. Our objective is to generalize the notion of conformable arrays of Fortran 90 and the semantics of assignments to undefined variables of MATLAB.

Let us first present three definitions that we will need in this section.

- We call *leaf elements* the elements of an HTA that do not have any components. The leaf elements could be empty containers or arrays of scalars. In the spirit of MATLAB, scalars cannot appear in isolation within HTAs and will be represented as 1×1 arrays.
- We say that an HTA is *complete* when all of its leaf elements are arrays of scalars. Otherwise, when some of the leaf elements are empty containers, the HTA is said to be *incomplete*. For example, arrays **`A`**, **`B`**, and **`C`** in the sequence (2.1) are complete. On the other hand, array **`A`** right after the sequence (2.2) is incomplete and will remain incomplete after the statements in (2.3) are executed, because these statements do not fill the containers **`A{1,2}`**, **`A{2,1}`** and **`A{2,2}`**.
- In Fortran 90, two arrays with the same shape (that is, the same number of dimensions and the same size in each dimension) are *conformable*. Also, scalars (and 1×1 arrays in our case) are conformable to arrays of any shape. Scalar binary operations such as add and multiply are extended in Fortran 90 to work on conformable objects. When both operands are arrays with the same shape, the operation is performed on corresponding pairs of scalars. When one of the objects is a scalar and the other an array, the scalar is operated with each of the elements of the array. Thus, **`c(1:10, 1:20:3)+d(1:10,1:7)`** is a valid Fortran 90 operation since the operands are both 10×7 arrays. Here, corresponding elements of the operands are added to each other to produce an array that is conformable to the operands. The expression **`e(:,:)+5`** is also valid and will add the scalar **`5`** to each element of array **`e`** producing a two-dimensional array with the shape of **`e`**.
- Two complete HTAs have the *same topology* if their outermost array of tiles have the same shape and corresponding outermost tiles are HTAs with the same topology or contain arrays of scalars that are conformable. This means that two HTAs will have the same topology if the only difference between them is on the leaves where the arrays have to be conformable, but do not have to have identical shapes.

We now proceed by discussing conformability, expressions, and assignment operations.

Conformability. Two complete HTAs are **conformable** if they *have the same topology or one of them is conformable to all elements at the top level of the other.* Notice that the second part of the definition is recursive. That is, if the smaller HTA does not have the same topology to one of the top level elements, then it must have the same topology of all components of this top level element, and so on. Informally, this definition means that two HTAs of different sizes will be conformable if the smaller one has the same topology of all elements of the other that are a certain level above the leaf elements. Notice also that our definitions implies that a scalar is conformable to any complete HTA.

Expressions involving HTAs. Following Fortran 90 the meaning of scalar operations is extended so that when the operands are both HTAs with the same topology the operation will be performed between corresponding scalar elements and will return an HTA with the topology of the operands. When the operands have different topologies, the smaller one is operated with all matching objects at the bottom of the hierarchy of the larger one. For example, adding a 2×3 array **`M`** is to HTA **`A`** resulting from sequence (2.1) is a valid operation that would result in **`M`** being added to all 2×3 arrays of scalars that are at the bottom of the hierarchy of **`A`**. Similarly, **`A + 3`** is valid and will add **`3`** to each scalar element in **`A`**. Notice that flattening changes the topology of an HTA. Thus, while the term **`B`** by itself represents the HTA computed in (2.1) and therefore has two levels of tiling, the term **`B{:,:}(:,:)`** represents an HTA with a single level of tiling. It is also possible to operate on a section of an HTA. Thus, **`B{1,:}{2:3}+1`** will operate on only one section of the HTA and will return an HTA with the shape of the section as illustrated in Fig. 5.

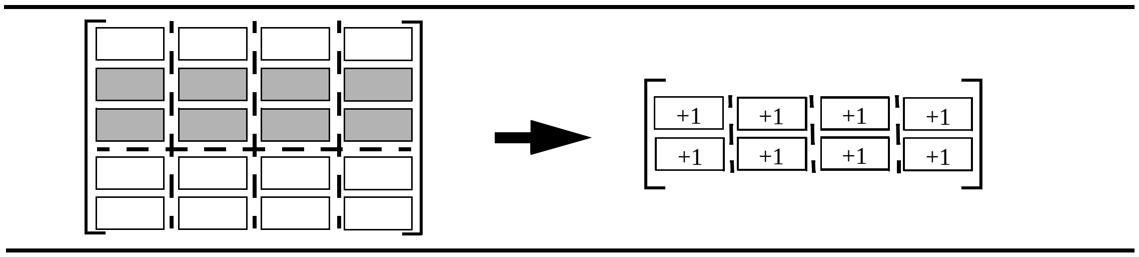

Figure 5. Operating on a section of an HTA.

Also following Fortran 90, scalar intrinsic functions are extended to operate on complete HTAs. These functions will operate on each scalar separately and will return an HTA with the topology of the operand. For example, **`sin(A)`** will return an HTA with the topology of **`A`**, but with each scalar replaced by its sine. Similarly, intrinsic array operations involving a single array will be extended in the natural way. For example, **`max(A)`** will return an HTA that will have the topology of **`A`**, except that every array of scalars will be replaced by a single scalar (which takes the form of a 1×1 array as stated above) which contains the maximum value of the array it replaces.

Assignments. Next, we generalize the semantics of assignment operations. In MATLAB, when the name of an array **`X`** appears in an expression, it refers to the whole array, but on the left hand side of an assignment statement **`X`** refers to the variable name as a container. Thus, in MATLAB if **`X`** is the one-dimensional array **`[1,2]`**, the expressions **`X+1`** and **`X(1:2)+1`** have the same meaning, adding one to each element. On the other hand, while **`X(1:2)=3`** will change **`X`** into **`[3,3]`**, **`X=3`** will change **`X`** into the scalar **`3`**. We extend this semantics to HTAs by assuming that references to containers that appear in expressions represent their content while on the left hand side of an assignment statement they represent the containers themselves. Thus, **`B=5`**, where **`B`** is the HTA constructed in (2.1) will replace **`B`** with the scalar **`5`**. However, **`B{:,:}{:,:}=5`** will replace each of the 2×3 arrays inside **`B`** by a 1×1 array containing **`5`** and **`B{:,:}{:,:}(:,:)=0`** will replace each of the 2×3 arrays inside **`B`** with a 2×3 array of zeros.

3.0 Mapping HTAs onto Memory

To specify how an HTA is to be mapped onto the memory of a machine, we could add a parameter to the functions for building HTAs introduced in the foregoing section or create a function variant for each type of mapping. We will follow the second approach in this paper.

We consider two classes of mappings. First, we discuss the mapping onto the main memory of a conventional uniprocessor or a shared memory multiprocessor. This mapping associates a unique memory location to each subscript value. Most programming languages assume a linear mapping, *M*, that for an *n*-dimensional array maps a particular value of the subscripts $(i_1, i_2, \ldots, i_n)$ onto a memory location as follows:

$$M(i_1, i_2, \ldots, i_n) = M_0 + \sum_{k=1}^{n} c_k i_k$$

The values of M_0 and c_k will depend on which dimension varies more rapidly as we traverse the consecutive locations where the array lies. Thus, in the row major order of the C language, for a two-dimensional array with shape $d_1 \times d_2$ containing elements of length *l*, the coefficients will be $c_1 = d_2\, l$, $c_2 = l$, and $M_0 = 0$ since the indices are assumed to start at 0. In the column major order of Fortran we have for the same array that $c_1 = l$, $c_2 = d_1\, l$ and $M_0 = (-d_1 - 1)\, l$, if the array indices are assumed to start at 1. The main advantage of the linear mapping is that computation of the memory location is simple and successive elements of an array along any dimension can be computed by addition without the need for multiplication. Compilers take advantage of this property via the strength reduction optimization.

Linear mapping can be done at any particular level of an HTA by laying out the tiles at this level in consecutive memory locations following a row major order or a column major order. We will assume that the functions **`tile`** and **`hta`** allocate objects in a row major order. To obtain column major order we would have to define new functions such as **`tileColumMajor`**, **`htaColumMajor`**, and **`randnColumMajor.`** Other layout functions that are advantageous for some classes of algorithms such as C-order, U-order, Hilbert order, and Z or Morton order can be attained similarly by creating the appropriate functions (e.g. **`tileCOrder`**, **`htaHilbertOrder`**) and extending the compiler to generate the corresponding address expressions.

Next, we discuss mapping onto the different nodes of a multicomputer or distributed-memory multiprocessor. To this end, we will assume that the nodes of the target machine form an *m*-dimensional mesh. The mesh (virtual) organization is by far the most frequently assumed topology in parallel programming. In our extension to MATLAB, we will use descriptors of node arrangement that is created by the **`nodes`** function and could be assigned to a variable. The invocation **`nodes(`**$d_1, d_2, \ldots, d_n$**`)`** returns a descriptor of a $d_1 \times d_2 \times \ldots \times d_n$ mesh of nodes.

For parallel programming, the top level of an HTA can be distributed across the nodes of a distributed memory machine using functions **htaD** and **tileD**. In the simplest case, when the top level of that HTA has the same shape as the node mesh, the meaning of the distribution operation is the obvious one: Each tile at the topmost level of the HTA is allocated to a different node. For example, to distribute HTA **A** created in (2.1) across a 2×2 node mesh we could modify the sequence as follows:

```
P = nodes(2,2);
C = tile(D,[2,4,6,8],[3,6,9]);
B = tile(C,[3],[1,2,3]);                    (3.1)
A = tileD(P, B,[1],[1]);
```

Here, tile **A{i,j}** is allocated to node (**i**,**j**), $1 \le$ **i** ≤ 2, $1 \le$ **j** ≤ 2. Similarly, to distribute the top level of HTA **G** created in (2.2), one tile per node, we could modify sequence (2.2) as shown next. Here, again tile **G{i,j}** is allocated to node (**i**,**j**), $1 \le$ **i** ≤ 2, $1 \le$ **j** ≤ 2.:

```
P = nodes(2,2);
G = htaD(P,2,2);
G{1,2} = hta(2,2);                          (3.2)
G{2,1} = hta(2,3);
G{2,2} = hta(3);
```

If the top level of the HTA has the same number of dimensions, but fewer components than the mesh of nodes where it is to be distributed, allocation will take place on consecutive processors starting at node 1 on each dimension. If the top level of the HTA has fewer dimensions than the mesh, then the top level HTA is extended with additional dimensions of size one to match the number of dimensions of the mesh. It is invalid for the top level of the HTA to have more dimensions than the mesh where it is to be distributed. If the top level of the HTA has more elements along one of the dimensions than the number of processors along that dimension, we assume a cyclical distribution.

4.0 Programming for Locality with HTAs

Following the pioneering work of McKellar and Coffman [McCo69], linear algebra computations are today usually organized to access arrays one tile at a time [ABDD92, Gust97, WhPD00]. The same approach has been studied as a compiler optimization technique where loops are automatically restructured so that arrays are accessed by tiles rather than in the more natural but less efficient row or column order [AbKL81,WoLa91, McCT96]. Although in some cases these algorithms require that the arrays to be manipulated be stored by tiles, in many cases this is not necessary and the reorganization of the computation usually suffices to significantly improve the memory hierarchy performance of the algorithms. Nevertheless, for large arrays, storage by tiles is desirable when the unit of transfer (page or cache line) is large [McCo69] or to avoid cache collisions [WhPD00].

The HTA notation of this paper should produce significantly more readable code when programming for locality. At the same time, our notation enables the layout of arrays in block order which should help performance for large arrays as was just mentioned. We now illustrate the benefit of HTA when programming for locality using the simple case of matrix multiplication. The typical matrix multiplication algorithm, implemented as a triply nested loop has the following form:

```
for i=1:n
      for j=1:n
            for k=1:n
                  c(i,j)=c(i,j)+a(i,k)*b(k,j);                    (4.1)
            end
      end
end
```

Here and in the following examples, we assume that **c** is initially all zeros. To improve the performance of this algorithm when the operand matrices are large, programmers or compilers can stripmine these loops and interchange them [PaWo96] to achieve the following code:

```
for I=1:q:n
      for J=1:q:n
            for K=1:q:n
                  for i=I:I+q-1
                        for j=J:J+q-1
                              for k=K:K+q-1
                                    c(i,j)=c(i,j)+a(i,k)*b(k,j);
                              end
                        end
            ...
end
```

This, loop is clearly much more complex than the original loop and would be even more complex had we not assumed that the size of the matrix, **n**, is a multiple of the block size, **q**. In contrast, the algorithm implemented on a tiled array stored as a single level HTA would have the following form:

```
for I=1:m
      for J=1:m
            for K=2:m
                  c{I,J} = c{I,J} + a{I,K}*b{K,J};              (4.3)
            end
      end
end
```

This is a much simpler and easier to read form of the same algorithm. One reason for the simplicity is the use of the HTA notation. It also helps that in MATLAB ***** stands for the matrix-matrix multiply operator. Notice that, for the algorithm to work not all tiles have to have the same size nor be square. Clearly, before code (4.3) executes, **a**, **b** and **c** must be created using functions such as **hta**, **tile** or one of their memory mapping variants.

Several levels of blocking can be useful in dealing with several levels of the memory hierarchy. The simplest way to extend the code in (4.3) to handle several levels of blocking is to replace **a{I,K}*b{K,J}** with an invocation to a user written function, say **matmul**, to get

```
            t = t + matmul(a{I,K},b{K,J});
```

The function would have a header of the form

```
                    function t=matmul(a,b)
```

and the body would the code in (4.3). Additional blocking levels could clearly be handled using recursion, that is by replacing **`a{I,K}*b{K,J} with matmul(a{I,K},b{K,J})`** in the body of the function. A function, **`ismatrix`**, that checks if its parameter is an array of scalars can be used to terminate the recursion as follows:

```
function t=matmul(a,b)
if ismatrix(a)
      t=a{I,K}*b{K,J});
      return
end
m=size(a,1);
t=hta(m,m);
t{:,:}=0;
for I=1:m
      for J=1:m
            for K=1:m
                  t{I,J} = t{I,J} + matmul(a{I,K},b{K,J});
            end
      end
end
```

5.0 Parallel Programming with HTAs

The only construct we use to express parallelism are array or collective operations on distributed HTAs. We assume that the main thread of our parallel programs will execute on a client sequential machine that could be a workstation. All variables, except distributed HTAs, will be assumed to reside in the memory of this client. The distributed HTAs on the other hand will be contained in the memory of a parallel server. All operations on elements of a distributed HTA will take place in the server as dictated by a simple version of the owner-computes rule: *the operations that compute values to be stored in an object must be performed in the node containing the object.* We assume that the compiler will take care of generating the code that is executed in the server and of inserting message-passing primitives so that the needed values are moved to the location before they are needed for the computation.

For example, assume that HTAs **`A`** and **`B`** have the topology of Fig 1(a) and that their tiles are distributed across a two-dimensional array of nodes or processors. One way to achieve this is to first allocate the top layer of **`A`** and **`B`** using he following sequence

```
                    P=nodes(5,4)
                    A=htaD(P,5,4)
                    B=htaD(P,5,4)
```

and then assign a 2×3 array of scalars to each tile.

Consider the statement

```
A{:,:}=A{:,:}.*B{:,:};
```

This statement means that for **1≤i≤5** and **1≤j≤4** tile **A{i,j}** and tile **B{i,j}**, should be multiplied element by element (**.*** is the element-by-element multiplication operator in MATLAB) and the result should replace tile **A{i,j}**. Since the result of multiplying **A{i,j}** by **B{i,j}** is to be stored in **A{i,j}**, the multiplication must take place in the node containing **A{i,j}**. Also, these multiplications can proceed in parallel with each other since the operation appears in an array statement. Notice that in this case no communication is necessary to execute the statement.

On the other hand, the statement

```
A{1:4,:}=A{1:4,:}.*B{2:5,:};
```

requires communication. Therefore, the compiler must generate the appropriate message-passing primitives so that for **1≤i≤4** and **1≤j≤4** tile **B{i+1,j}** be copied to a temporary in the node containing **A{i,j}** before the operation can take place.

Consider finally the statement

```
A{:,:}=A{:,:}.*X;
```

where **X** is a variable residing in the client. In this case the compiler will have to generate a broadcast operation to send the value of **X** to all nodes before the operation can take place.

Before proceeding with the examples, we need to make an additional extension to MATLAB. As mentioned above, in MATLAB when the operands are arrays, the ***** operator represents matrix multiplication and **.*** represents element by element multiplication. With the introduction of tiled arrays, we introduce additional level in the data hierarchy and the meaning of ***** and **.*** must be extended. We will assume that ***** between to HTAs with the topology of Fig 1(a) will produce the effect of matrix multiplication at the tile level. Thus, we will assume that **c{:,:}=a{:,:}*b{:,:}** or simply **c=a*b** will have the same effect as loop 4.3. If we just wanted to multiply corresponding submatrices, we will write **c{:,:}=a{:,:}.*b{:,:}** or **c=c.*b**. This will be equivalent to the loop

```
for I=1:m
      for J=1:m;
            c{I,J}=a{I,J}*b{I,J};
      end
end
```

Notice that in the loop the operands of ***** are matrices and therefore the operator stands for matrix multiplication, the same meaning it has in MATLAB. Finally, if we just want to multiply corresponding scalars in two-level HTAs, we would write **c=a..*b.**

Next, we present two examples of parallel programs using HTAs. The first is a dense matrix-matrix multiply and the second is a matrix vector multiply where both the matrix and the vector are sparse.

For our first example, we will implement the SUMMA algorithm. The algorithm has a very simple representation using HTAs. To explain the algorithm, consider first the matrix multiplication loop (4.3) with the innermost loop (loop **`K`**) moved to the outermost location (recall that we assume **`c`** is initialized to zero)

```
for K=1:m
     for I=1:m;
          for J=1:m
               c{I,J} = c{I,J} + a{I,K}*b{K,J};
          end
     end
end
```

The inner two loops increment the array **`c{:,:}`** so that for **`1≤I≤m`** and **`1≤J≤m`** tile **`c{I,J}`** is incremented by **`a{I,K}*b{K,J}`** on each iteration of the outermost loop. Notice that for each **`I`**, **`J`**, and **`K`**, the tiles **`a{I,K}`** and **`b{K,J}`** are each used in the computation of **`m`** different tiles of **`c`**. Also, the inner two loops are a parallel operation on the two-dimensional array of tiles **`c`** which can be easily represented in array form if **`a`** and **`b`** are appropriately replicated. The introduction of the replication operations, leads directly to the SUMMA algorithm.

In our notation, we can achieve the replication using a function that we call **`tileSpread`**. This function has the same semantics as the Fortran 90 **`spread`** function except that **`tileSpread`** operates on distributed arrays of tiles instead of on arrays of scalars like **`spread`**. We will use **`tileSpread`** to create a two-dimensional array of tiles from a one-dimensional array of tiles. The first parameter is the one-dimensional array to be replicated. The second parameter determines whether the one-dimensional array will be replicated as columns or as rows. The value of the second parameter will be 2 in the first case and 1 in the second. The third parameter determines how many copies to create.

```
for K=1:m
     t1{:,:}=tileSpread(a{:,K},2,m);
     t2{:,:}=tileSpread(b{K,:},1,m);
     c{:,:}=c{:,:}+t1{:,:}.*t2{:,:};
end
```

The **`tileSpread`** function when applied to distributed HTAs could be implemented in many different ways depending on the characteristics of the target machine and the mapping of the source HTA onto the parallel machine.

The previous loop can be written in a simpler form:

```
for K=1:m
     c{:,:} = c{:,:} + a{:,K}*b{K,:};
end
```

This representation leaves the decision of how to implement the broadcasting of **`a`** and **`b`** to the compiler while in the previous loop the programmer exercises some control by choosing the appropriate routine. It is of course possible to have even more control over the way the broadcast is done if desired. Thus, the following code sequence achieves the same goal as the statement **t1{:,:}=tileSpread(a{:,K},2,m);**:

```
t1{:,K}=a{:,K};
for L=K+1:m
      t1{:,L}=t1{:,L-1};
end
for L=K-1:-1:1
      t1{:,L}=t1{:,L+1};
end
```

The second example will be a matrix vector multiplication where both the vector and the matrix are sparse. Coding is significantly simplified by the way MATLAB handles sparse computations. In fact, sparse matrices are operated in MATLAB using the same syntax used for dense computations. The MATLAB interpreter automatically selects the appropriate procedure to handle sparse data.

Let us assume first that the data is originally in matrix **a** and vector **b**, both located in the client. Array **a** will be distributed by blocks of rows across the nodes of the target machine. To this end, **a** is assigned to HTA **c** that is just a distributed linear arrangement of containers. Also, vector **b** will be distributed by blocks of elements using HTA **v**. There is a vector **dista** in the client that specifies which rows of **a** are to be assigned to **c{I}**. These are rows **dista(I)** to **dista(I+1)-1**. Similarly, array **distb** specifies which elements of **b** will be assigned to **v{I}**. The first step of the code would, therefore, contains the following statements:

```
Step 1:     P=node(n);
            c=htaD(P,n);
            v=htaD(P,n);
            for I=1:n
                  c{I}=a(dista(I):dista(I+1)-1),:);
                  v{I}(distb(I):distb(I+1)-1)=b(distb(I):distb(I+1)-1);
            end
```

If matrix **a** or vector **b** are too large to fit the client, the previous loop could be easily replaced by an I/O function that will read the data directly to the components of **c** and **v**.

The matrix vector multiplication will be performed in chunks. In fact, each node, **I**, will compute a chunk of the vector by multiplying **c{I}** by **v**. However, only the elements of the vector corresponding to nonzero columns of **c{I}** are needed. If we provide to each node a copy of vector **distb**, by analyzing **c{I}** and correlating the result with **distb** the node can easily determine, for each **J,** which elements of **v{J}** will be needed to perform the **c{I}*v** operation. The result of this analysis will be stored in HTA **w**. Node **I** will assign to each **w{I}{J}** a vector containing the indices of the elements needed from **v{J}**. We assume the existence of a function **need** that computes **w{I}.** This function should be asy to write by a programmer familiar with MATLAB. The second step of our algorithms is then to call the function **need** as follows:

```
Step 2:     forall I=1:n
                  w{I}=need(c{I},distb)
            end
```

Here, we have used the **`forall`** construct with the same meaning it has in Fortran 90. MATLAB does not have such a construct, but we have found it necessary in many cases to implement parallel algorithms.

The next step in the algorithm is to send the data in **`w{I}{J}`** (contained in node **`I)`** to node **`J`** for all **`I`** and **`J`**. In this way, node **`J`** will know which elements of its vector block, **`v{J}`**, are needed by node **`I`**. We will store this information in HTA **`x`** so that **`x{I}{J}`** will contain a vector of the indices of the elements needed by node **`J`** from node **`I`**. Clearly, **`x`** is the transpose of **`w`**, therefore step 3 of the computation will just be:

```
Step 3:      x=tileTranspose(w);
```

In the next step each node, **`I`**, gathers, for all **`J`**, into **`y{I}{J}`** the elements of **`v{I}`** needed by node **`J`**. Then this data is sent to the appropriate node using another transpose operation:

```
Step 4:      forall I=1:n
                   for j=1:n
                         y{I}{J} = v{I}(x{I}{J}(:));
                   end
             end
             z=tileTranspose(y);
```

Finally, each local vector is extended with the data that just arrived into **`z`** and the matrix vector multiplication can be performed:

```
Step 5:      forall I=1:n
                   v{I}(x{I}(:)) = z{I}(:);
             end
             v{:}=c{:}*v{:};
```

6.0 Conclusions

The parallel programming approaches that have attracted most attention in the recent past fall at the two extremes of the range of possible designs. On one hand there is the SPMD, message-passing programming model. MPI is by far the most popular implementation of this model, but not the only one. Two parallel programming languages, Co-Array Fortran and UPC, are other examples of this model. The incorporation of communication primitives into programming languages like Co-Array Fortran and UPC significantly reduces the amount of detail that must be specified for each communication operation in the library-based approach of MPI. However, in our opinion, Co-Array Fortran and UPC do not go far enough due to their adoption of the SPMD model which can easily lead to unstructured code. This lack of structure could be the result of communication taking place between widely separated sections of code with the additional complication that a given communication statement could interact with several different statements during the execution of a program. In theory at least, the lack of structure possible with SPMD programs could be much worse than anything possible with the use of **`goto`** statements in conventional programming. In other, perhaps more colorful, words what we are saying is that the use of the SPMD programming model could lead to *four-dimensional spagetti code*.

The other class of parallel programming models on the spotlight mostly follows a single-threaded model. Languages in this class include the OpenMP directives [CDKM00] and High Performance Fortran. One difficulty with OpenMP is that it assumes a shared memory support that is not always available in the hardware of today's machine.The shared-memory model could be implemented in software, but that often leads to highly inefficient parallel programs. A second, and much more serious, limitation of OpenMP is that the directives do not explicitly represent the notion of locality. This is a very important notion for parallel programming since distributed memory is a physical necessity in large-scale multiprocessors. It could be said that the compiler could take care of rearranging the code and distributing data to take care of locality, but a proven compiler technology for this purpose is not at hand today. High-Performance Fortran is based on sequential source code complemented with directives mainly for specifying how data is to be distributed. It is the task of the compiler to transform the sequential code into SPMD form and generate all the necessary communication primitives. Unfortunately, from the poor reception given to HPF it seems that automatically producing highly efficient code from HPF source is beyond the capabilities of today's technology.

Our proposal lies somewhere between these two extremes. The programming model is single-threaded, but communication and distribution is explicit. Therefore, the requirements from the compiler should be more modest that those of HPF. Our experience in the programing of both dense and sparse kernels is that the use of array notation and the incorporation of tiling in a native data type significantly improve readability when programming for locality and parallelism. This is clearly due to the importance of tiling for parallel programming and for locality, a fact that has become increasingly evident in the recent past.

7.0 References

[AbKL81] W. A. Abu-Sufah, D. J. Kuck, D. H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. **IEEE Transactions on Computers** 30(5): 341-356(1981).

[ABDD92]E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[CDCY99]W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification*
CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[CDKM00]R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. Parallel Programming in OpenMP.Morgan Kaufmann; (October 2000)

[GrES99] W. Gropp, L. Ewing and A. Skjellum. **Using MPI,** The MIT Press, 2nd Edition, 1999.

[Gust97] F. G. Gustavson, *Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms*, **IBM J. Res. & Dev. 41,** No. 6, 737–755 (November 1997).

[HiKT92] S. Hiranandani, K. Kennedy and C. Tseng *Compiling Fortran D for MIMD Distributed--memory Machines.* **CACM**, 35(8):66-80, August 1992.

[Iver62] K.E. Iverson, A Programming Language. Wiley, 1962.

[KoMe92] C. Koelbel and P. Mehrotra. *An Overview of High Performance Fortran*, **ACM SIGPLAN FORTRAN Forum**, Vol. 11, No. 4, December, 1992.

[PaWo86] D. Padua and W. Wolfe. *Advanced Compiler Optimizations for Supercomputers*. Communications of the ACM, 29(12), 1986.

[McCo69] A. C. McKellar and Edward G. Coffman Jr.: *Organizing Matrices and Matrix Operations for Paged Memory Systems.* **CACM** 12(3): 153-165 (1969)

[McCT96] K. S. McKinley, S. Carr, and C.-W. Tseng. *Improving data locality with loop transformations*. ACM Transactions on Programming Languages and Systems, 18(4):424-453, July 1996.

[NuRe98] R. W. Numrich and J, Reid. *Co-Array Fortran for parallel programming*. **ACM SIGPLAN FORTRAN Forum,** Vol. 17 (1998) 1--31

[WhPD00] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. *Automated empirical optimizations of software and the atlas project*. Available as http://www.netlib.org/lapack/lawns/lawn147.ps, 19 September 2000.

[WoLa91] M. E. Wolf and M. S. Lam. *A data locality optimizing algorithm*. SIGPLAN Notices, 26(6):30--44, June 1991. **Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.**