Hierarchically Tiled Arrays for Parallelism and Locality [*]

Jia Guo, Ganesh Bikshandi, Daniel Hoeflinger,
Gheorghe Almasi[†], Basilio Fraguela[‡], María Jesús Garzarán, David Padua, and Christoph von Praun[†]

University of Illinois at Urbana-Champaign
{jiaguo, bikshand, hoefling, garzaran, padua}@cs.uiuc.edu

[†]IBM T.J. Watson Research Center
Yorktown Heights
gheorghe, praun@us.ibm.com

[‡]Universidade da Coruña,
Spain
basilio@udc.es

Abstract

Parallel programming is facilitated by constructs which, unlike the widely used SPMD paradigm, provide programmers with a global view of the code and data structures. These constructs could be compiler directives containing information about data and task distribution, language extensions specifically designed for parallel computation, or classes that encapsulate parallelism. In this paper, we describe a class developed at Illinois and its MATLAB implementation. This class can be used to conveniently express both parallelism and locality. A C++ implementation is now underway. Its characteristics will be reported in a future paper. We have implemented most of the NAS benchmarks using our HTA MATLAB extensions and found during that HTAs enable the fast prototyping of parallel algorithms and produce programs that are easy to understand and maintain.

1 Introduction

This paper describes a class of objects, hierarchically tiled arrays (HTAs) [4], which can be used to represent both tiled parallel computations and sequential computations tiled for locality enhancement. Hierarchically tiled arrays are a natural extension of the array type of Fortran 90 and MATLAB™ and, as a result, can be used to write very readable parallel code in these languages.

The HTA class is the result of our efforts to develop a parallel implementation of MATLAB™. Although there have been many MATLAB™ extension proposals [6, 13, 20], none, seemed to us, enabled a natural representation of parallel algorithms. We decided to use an extension of a MATLAB™ object called cell array, to represent both sequential and parallel tiled computations. This led to the design of hierarchically tiled arrays. However, although HTAs were inspired by a MATLAB construct and its first implementation was in MATLAB, HTAs can be implemented in any OO language. In fact, a C++ implementation is now underway.

In simple terms, hierarchically tiled arrays (HTAs) are arrays whose elements are tiles. Tiles contain either conventional arrays or lower level hierarchically tiled arrays. The main motivation behind the design of HTAs is that, for many algorithms and program implementations, tiles have proven convenient to attain locality in sequential computations [21] and represent data distribution and communication in parallel algorithms [2, 8, 10, 18]. To represent parallel computations, the highest level tiles of a HTA are distributed across the processors of a parallel machine. Communication can then be represented as assignments between distributed HTAs and parallel computation as operations on the distributed tiles.

HTAs are intended for *explicit* parallel programming and, as a result, HTA programs are more complex than implicitly parallel codes implemented in languages such as HPF. This is the price that must be paid for an environment that does not require sophisticated compiler algorithms and give the programmer direct control over parallelism and communications. On the other hand, codes based on HTAs are higher level than MPI codes and, as will be seen in the examples presented in this paper, the representation of parallel operations as array assignments and expressions imposes an structure on parallel programs similar to that imposed on conventional codes where `goto` loops are replaced with structured constructs like `for` and `while` loops. Thus, HTAs force structured parallel programs which are better in terms of productivity than the more unstructured, or assembly-like, forms that can arise when using message passing libraries like MPI.

HTAs can be implemented by extending the language, which would require modifying the compiler, or by implementing a new class in an object-oriented language. For the work reported in this paper we followed the later approach and used the OO capabilities of MATLAB™ to implement HTAs. Thanks to *operator overloading*, the syntax of HTA operations in MATLAB™ is as compact and readable as the syntax of conventional array operations.

Similar to other programming systems, like pC++ [5], that encapsulate parallelism in method invocations, HTA operations are invoked from a single control thread, which in our current implementation is a conventional MATLAB™ program. Conceptually, we could think of the control thread as executing on a (serial) workstation that takes care of all non-HTA operation including user interaction. The HTAs can be assumed to be stored in a parallel co-processor that execute operations on these objects. If we view programs this way, we avoid the potential problems of SPMD programming, where explicit communications could be

[*]This material is based upon work supported by the National Science Foundation (NGS program) under Grant No. 0103610. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

mixed with computations in an unstructured manner which in the worst case might lead to, difficult to understand, four-dimensional spaghetti code.

Although it is easy to reason about HTA programs in terms of the workstation/co-processor model, such implementation is inefficient. The overhead of broadcasting computation requests and data from the workstation to the processors in the server would hinder performance. For that reason, our implementation follows the SPMD model. The translation from the workstation/co-processor model to the SPMD model is trivially attained by (1) replicating the control thread and all non-HTA data on each processor of the parallel machine and (2) implementing HTA operations or methods in work-sharing fashion so that each processor carries out a part of the computation. Thus the system can take advantage of all the performance benefits of SPMD without any of its productivity drawbacks.

The rest of this paper is organized as follows. In Section 2, the HTA data type, including its structure and operations, is discussed. Then, Section 3 presents the main ideas behind our SPMD implementation. Section 4 presents our HTA programming approach using the NAS benchmarks and Section 5 compares HTAs with other parallel programming paradigms. Finally, Section 6 presents our conclusions.

2 The HTA Programming Paradigm

In this section, we first describe the syntax and semantics of the Hierarchically Tiled Array (HTA) [4] (Section 2.1) and then outline the main characteristics of our approach (Section 2.2).

2.1 Syntax and Semantics

We define a *tiled array* as an array partitioned into tiles in such a way that adjacent tiles have the same size along the dimension of adjacency. A *hierarchically tiled array* (HTA) is a tiled array where each tile is either an unpartitioned array or an HTA. The tiles can be distributed across processors in a distributed-memory machine. HTAs can be used to facilitate the expression of both locality and parallelism. We distribute the outermost tiles across processors for parallelism and utilize the inner tiles to implement computations that exhibit high locality. Figure 1-(a) shows an example of an HTA with two levels of tiling.

HTA operators in our MATLAB^{TM} implementation overloads the standard operators of the language so that HTA storage and computations can be distributed across the processors of a parallel system.

2.1.1 Construction of HTAs

An HTA can be created as an empty set of tiles, which we call empty HTA. Alternatively, an HTA can be the result of tiling an existing array. Also, HTAs may be local or distributed.

In order to create an empty HTA, the HTA constructor is called with the number of desired tiles per dimension. For example,

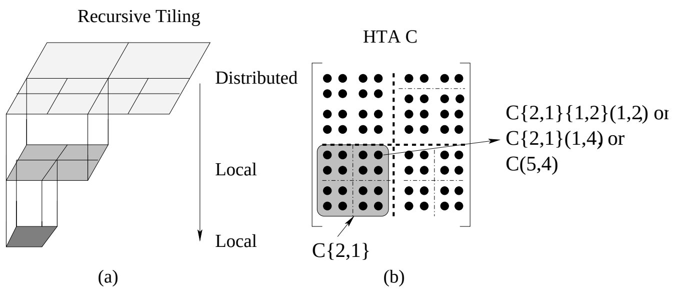


Figure 1: Pictorial view of a hierarchically tiled array.

`hta(3, 3)` generates an HTA with 3×3 empty tiles. To complete the HTA, each tile must be assigned a content after the empty shell is created. Alternatively, an HTA can be built by partitioning an array with a series of delimiters in each dimension. For example, if `M` is a 6×6 matrix, the function `hta(M, {[1 3 5],[1 3 5]})` creates a 3×3 HTA resulting from partitioning `M` in tiles of 2×2 elements each. The second parameter of this function is an array of vectors that specifies the starting location of the tiles. The i-th vector contains the *partition vector* for the i-th dimension of the source array. The elements of this partition vector mark the beginning of each sub-tile along the corresponding dimension. In our example rows 1 , 3 and 5, and columns 1, 3 and 5 are the partitions points as shown in Figure 2-(a).

To distribute the tiles across a processor, the constructor must receive as argument a vector defining the dimensions of the mesh. This way, `hta(3, 3, [2, 2])` generates an empty 3×3 HTA distributed on a 2×2 processor mesh. Figure 2 shows an example where a 6×6 matrix is distributed on a 2×2 mesh of processors, as the last parameter of the HTA constructor indicates. In our current implementation, the default distribution is a block cyclic distribution of the matrix contained in the HTA, with the blocks defined by the topmost tiling. Other distributions can be specified by adding as one of the parameters to the HTA constructor the name of the function that specifies the mapping. For example, `hta(3, 3, [2, 2], @func)`[1], will apply the mapping specified by the function `func` to the HTA to be distributed on a 2×2 mesh of processors.

Finally, although not shown here because of space limitations, HTAs can be built with several levels of tiling, like those shown in Figure 1-(a).

2.1.2 Accessing the Contents

References to HTAs allow access to both tiles and scalar elements. Curly brackets are used when we index tiles, while parenthesis denote the access to scalar elements within the HTA or its tiles.

Figure 1-(b) shows some examples. The expression `C{2,1}` refers to the lower left tile. Also, the element in the fifth row and fourth column can be referenced using `C(5,4)`, just as if `C` were

[1]In MATLAB^{TM} the character `@` before a function name creates a pointer to that function.

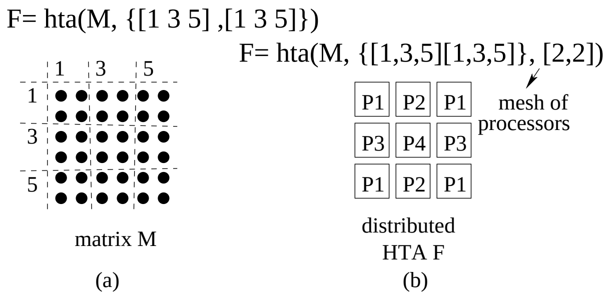


Figure 2: Construction of an HTA by partitioning an array-(a). Mapping of tiles to processors-(b)

an unpartitioned array. This element can also be accessed by selecting the bottom-level tile that contains it and its relative position within this tile: `C{2,1}{1,2}(1,2)`. A third expression representing `C(5,4)` selects the top-level tile `C{2,1}` that contains the element and then *flatten*s or disregards its internal tiled structure: `C{2,1}(1,4)`.

2.1.3 Binary Operations and Assignments

We generalize the notion of conformability of Fortran 90 for use on HTA operations. When two HTAs are used in an expression, they must be conformable. That is, they must have the same topology (number of levels and shape of each level), and the corresponding tiles in the topology must have sizes that allow to operate them. In fact, the operation actually takes place tile by tile, and the output HTA has the same topology as the operands.

An HTA can also be conformable to an array and it is always conformable to a scalar. In the first case, the array is operated with each one of the lowest-level tiles of the HTA, provided that the tiles and the array are conformable in the Fortran 90 sense. That is, they have the same shape. When the other operand is a scalar, it is operated with each scalar component of the HTA. Again, the output HTA has the same topology as the input HTA.

The rules for assignments to HTAs are similar to those regulating binary operations. When a scalar is assigned to a range of positions within an HTA, the scalar is replicated in all of them. When an array is assigned to a range of tiles of an HTA, the array is replicated to create tiles. Finally, an HTA can be assigned to another HTA (or a range of tiles of it).

2.2 A Paradigm with a Global Unified View

The HTA programming model corresponds to that of a global view language where the programmer specifies the overall behavior of the algorithm rather than focusing on the behavior on a per processor basis. Data are also global and are handled in a unified way. The HTA programming model provides a deterministic semantics: each statement is completed before the next one begins its execution; and the right hand side of assignments is evaluated before assignment takes place. As a result, programmers need not specify synchronization or be concerned with deadlocks or race conditions. Thus, HTA syntax and semantics are simpler and cleaner than those of SPMD approaches. The downside is that asynchronous overlap cannot be explicitly stated using the current HTA operations, but much of this overlap can be achieved automatically with the appropriate implementation (see last paragraph of Section 3).

HTAs improve programmers' productivity, since they can use familiar programming languages and sequential modules, perhaps with small changes. So, programmers can write parallel programs the same way they write sequential programs and they can gradually migrate sequential applications to parallel forms.

HTA syntax makes communication explicit, since computations involving tiles located in different processors will result in tiles movement. This provides a simple, but powerful performance model. This is something missing from HPF [12, 15] and other similar global view parallel programming approaches.

3 Implementation

HTAs can be added to any object-based or object-oriented language. We chose the MATLAB^{TM} environment as the host for our first implementation because it is a linear algebra language with a large base of users who write scientific code. Also, MATLAB is an object oriented language designed to be extensible. Third party developers can provide so-called *toolboxes* of functions for specialized purposes.

We wrote a toolbox that contains the HTA class with its methods, such as `hta` (the constructor of the class), `circshift`, and other methods that overload standard MATLAB^{TM} operators and functions. Methods that do not involve communication were written in MATLAB^{TM} to simplify code development. Small methods used very frequently were written in C for performance reasons. The communication between the processors is implemented using MPI [11]. Our framework requires that all the processors participating in the system have a copy of MATLAB^{TM} and the HTA toolbox. Since MATLAB^{TM}is interpreted, our library-based implementation suffers from the overheads that a compiler could easily remove [3, 19].

Our approach presents programmers a global view of the code. However, to implement our toolbox, the resulting code follows the SPMD execution model for efficiency. In our implementation, all processors execute the same program. Scalar variables, arrays and non-distributed HTAs are replicated in each processor. Distributed HTAs are also created in every processor, even in those that do not own tiles of the HTA. In this latter case, processors will only keep information about the structure of the HTA. When a unary operation on a distributed HTA is executed, each processor acts on the tiles of the HTA it owns. In the case of binary operations, the tiles of the right operand not co-located with the tiles of the left operand are copied (using MPI primitives). Then, the processors that own the tiles of the left operand perform the desired operation and store the resulting tiles. In this way, the HTA resulting from a binary operation has always the same mapping as that of the left operand. Of course, when the HTAs to operate are mapped in the same way, there is no communication. Similar communication occurs when the right hand side (RHS) of an assignment is not mapped in the same way as the left hand side (LHS). Every tile of the RHS HTA of the assignment is copied to the processor that owns the corresponding tile of the LHS HTA to which it is assigned. With these simple rules, the programmers can know at all times where and how communication takes place.

Our implementation also allows concurrent execution of statements without affecting the sequential deterministic semantics of our approach. When a processor does not own any of the tiles in the LHS or RHS of the computation, the statement in this processor is equivalent to *no-op*. Those processors can proceed to the next statement without waiting for the other processors to finish. Thus, the overlapping of different statements is possible.

4 NAS Benchmarks

The NAS benchmarks are a set of programs that were designed to evaluate the performance of parallel supercomputers [1]. We have implemented six of the NAS benchmarks (EP, MG, CG, FT, BT, and LU) using HTAs. For each benchmark we wrote a serial version using plain MATLAB™and a parallel version using MATLAB™+ HTAs. In each case our implementation follows the same strategy as the public-domain MPI implementation of the NAS benchmarks [1]. Section 4.1 to Section 4.6 discuss the main characteristics of HTA implementation for each benchmark. In Section 4.7, we will compare our approach with MPI programming style.

4.1 EP

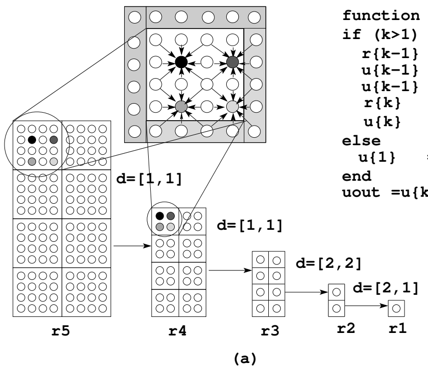
EP is an Embarrassingly Parallel benchmark. It can be used to determine the peak parallel performance of a machine. EP generates pairs of Gaussian random deviates according to the scheme described in [14]. EP is fully parallel, except at the end of the application, where a sum reduction across processors is performed. EP can be written using HTAs with a special method `feval` as shown in Figure 3.

`feval` applies a function, which may be user-defined, to each tile of one or several input HTA(s). A call to this method has the form `feval(@func, arg1, arg2, ...)`, where `@func` is a pointer to the function to execute, and `arg1`, `arg2`,... are the arguments for its execution. At least one of these arguments must be an HTA. In EP, the procedure to perform the necessary computation (`ep`) is applied to each of the tiles of the input HTA.

`reduceHTA` is a generalized reduction method that operates on HTA tiles. It has the form `reduceHTA(@func, h, dim, isAlltoAll)` where `@func` is a pointer to the reduction function, `h` is the HTA target of the reduction, `dim` is the dimension of reduction with 0 representing reduction along all dimensions, and the boolean `isAlltoAll` specifies whether it is an all-to-all reduction. In Figure 3, suppose `q` is a 2×2 HTA, `reduceHTA(@sum, q, 2, false)` returns a column HTA of size 2×1 where the upper tile contains the sum of the values in the upper tiles, and the lower tile contains the sum of the values of the lower ones. The resulting column HTA maps to processors containing the tiles in the first column of HTA `h`.

```
q = feval(@ep, in);
r = reduceHTA(@sum, q, 2, false );
sx = reduceHTA(@sum, sx, 0, false );
sy = reduceHTA(@sum, sy, 0, false );
```

Figure 3: HTA implementation of EP



```
function [uout] = mgrid(r,u,k)
if (k>1)
  r{k-1} = project(r{k}, r{k-1})
  u{k-1} = mgrid(r,u,k-1)
  u{k-1} = interp(u{k-1},u{k})
   r{k}   = resid(u{k},r{k})
   u{k}   = psinv(r{k},u{k})
else
  u{1}   = psinv({r{1},u{1})
end
uout =u{k}
```

(b)

Figure 4: MG Benchmark. (a) Pictorial view. (b) Code for MG.

4.2 MG

MG solves the 3D Poisson's equation, $\nabla^2 u = v$, using the finite-difference method. The system of equations resulting from the finite-difference method is solved using the multigrid V-cycle algorithm. The major steps in the algorithm are the two inter-grid operations `projection` and `interpolation`. `Projection` projects the residual value from a grid to a coarser grid. `Interpolation` interpolates the error solution from a grid back to a finer grid. The various levels of grids are represented using an array of HTAs, where each HTA h_k has half as many elements as HTA h_{k+1}. The core computation in MG is the `stencil` computation.

A pictorial view of the `projection` function is shown in Figure 4-(a) for a 2D grid. Each highlighted grid point in HTA r_4 in Figure 4-(a) is computed as the weighted average of the corresponding highlighted grid point and all its neighbors in HTA r_5, as shown in the zoomed HTA, where the shadow regions to allocate the data from the neighbors HTA are also shown. Figure 4-(b) shows the code for MG implemented as a recursive function. For each grid at level k, the `projection` function is called until it reaches the grid at level 1. At this point the recursion ends and the `interpolation`, `resid` and `psinv` functions are called on increasingly finer grids.

4.3 CG

CG uses inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of zeros. The core of CG consists of a 2-D sparse matrix-vector multiplication, three saxpy operations and two vector additions. For lack of space, we only describe here the matrix-vector multiplication shown in Figure 5

```
A = hta(MX,{partition_A}, [M N]);
B = repmat(hta(x,{partition_B},[1 N]),[M 1]);
C = A * B;
y = reduceHTA(@sum, C, 2, true);
```

Figure 5: HTA code for sparse matrix-vector multiplication

The implementation of this benchmark is greatly simplified by MATLAB^{TM}'s support for sparse computations. The objective of the code in Figure 5 is to multiply sparse matrix MX by a vector x. The matrix MX is tiled along both dimensions and distributed on a $M \times N$ mesh of processors. The result is stored in A.Vector x is replicated to create a copy on each row of processors and is partitioned along each row of processors to conform to the horizontal partitioning of A. The array consisting of the copies of x is called B. To obtain the result of multiplying A by x, first B is multiplied by A on a per-tile basis. This is followed by a all-to-all sum reduction of the result across the columns and vector y receives the final result.

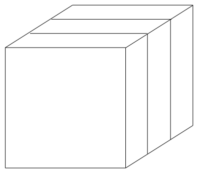
4.4 FT

FT solves partial differential equations (PDE) using forward and inverse Fourier transform (FT). The parallel Fourier transform for an n-dimensional array of size $(N_1 \times N_2 \cdots \times N_n)$, decomposes the array into tiles of size $(N_1 \times \frac{N_2}{d_2} \cdots \times \frac{N_n}{d_n})$, where $d_i \geq 1$. FT along the first dimension (unpartitioned) is just a local FT operation. To compute the FT along dimension i where $d_i > 1$, the $i - th$ dimension is transposed with the first dimension and the local FT is applied along the first dimension.

Figure 6-(b) shows an outline of our implementation of this algorithm for a 3-D array where only the third dimension of the HTA u is distributed, as shown in Figure 6-(a). FT along the first and second dimension of an HTA is computed using the overloaded version of the standard MATLAB^{TM}fft operator which applies the standard MATLAB^{TM} fft to each of the tiles of the HTA along the dimension specified in the third parameter. To apply the fft along the third dimension, we need to make this dimension local to a processor. For that, we transpose the HTA using the HTA dpermute operator. The dpermute operation transposes the data, but the shape of the containing HTA remains constant.

4.5 BT

BT solves the Navier-Stoke's equation for 3-D grids using Alternating-Direction Implicit (ADI) method. Finite-difference approximation on a 3D grid leads to a 7-diagonal banded system.



```
u = fft (u, [],1);
u = fft (u, [],2);
u = dpermute( u,[3 1 2]);
u = fft( u, [],1);
```

(b)

Figure 6: FT Benchmark.(a) Pictorial View. (b) HTA code.

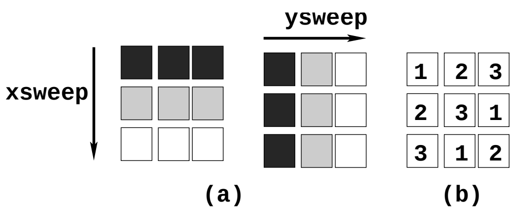


Figure 7: BT Benchmark. (a) Flow of Computation. (b) Processor Mapping.

To solve such a system, the implicit 3-D finite difference equation is split into three separate equations. The results are tri-diagonal systems which are solved using Gaussian elimination. This leads to three sweeps in different directions - first across the x dimension (xsweep), the second along the y dimension (ysweep) and the third along the z dimension (zsweep). The flow of computation in the HTA program for a 2D case is shown in Figure 7-(a), where xsweep and ysweep are shown. In each case, tiles of the same color can be operated at the same time. Thus, the key to achieve parallelism in this benchmark is to have multiple tiles assigned to a processor and a cyclic mapping of processors as shown in Figure 7-(b). Such a mapping ensures that during each sweep, all the processors are working in parallel.

4.6 LU

LU solves the Navier-Stoke's equation for 3-D rectangular grids. LU uses Symmetric Successive Over Relaxation (SSOR) algorithm to solve the problem, which at the end is resolved by forming the sub-block and super-block diagonal matrices (carried out by methods jacld and jacu) and solving the lower and upper triangular systems (carried out by methods blts and buts). Methods jacld and jacu are completely data parallel, while blts has dependences where the grid point (i,j,k) depends on points $(i-1,j,k)$, $(i,j-1,k)$ and $(i,j,k-1)$. In buts the grid point (i,j,k) depends on grid points $(i+1,j,k)$, $(i,j+1,k)$ and $(i,j,k+1)$. The computation can be carried out in parallel following a wavefront strategy [16].

A parallel wavefront computation appears when the value of an element depends on the value or values of neighboring elements computed in previous iterations. These codes can be efficiently parallelized by computing in parallel the element of each "diagonal" of the matrix, where the angle of the diagonal is a function of the dependences [16]. The processors compute local data before sending them to the processors containing the dependent data. Similarly, wavefront computations can be parallelized in tiled fashion. Figure 8-(a) shows a serial 2D wavefront computation. The tiled, HTA version is shown in Figure 8-(c), where logical indexing is used to determine the tiles that can operate in each iteration of the K loop. The expression I+J==K composes a boolean array. Tiles with true value in the boolean array will be selected to do the 2D wavefront computation.

A pictorial view of how the computation advances across tiles is shown in Figure 8-(b), where the values of the I and J matrices are also shown. In the Figure, a is a $M \times N$ HTA, distributed on a $M \times 1$ processor mesh, so that each row of tiles is mapped to the same processor. The last two statements in Figure 8-(c) copy the last row and column of the tiles of the K-th diagonal to the first row and column of the corresponding tiles in the K+1-th diagonal. This is done to initialize the *shadow* of these tiles which guarantee that all the data needed for each computation is in the same tile.

Our implementation of the NAS LU benchmark is similar to the code shown in the Figure 8 using logical indexing. The only difference is that Figure 8 shows a 2-D wavefront, while the NAS LU benchmark is a 3-D wavefront and, as a result, the execution advances through a hyper-plane instead of a diagonal line. In LU data are partitioned into $M \times N \times K$ tiles, and distributed across $M \times N$ processors, so that the third dimension is local to each processor. The code in Figure 8 corresponds to blts, while the code for buts would be similar but in the opposite direction.

4.7 Summary and Comparison with MPI codes

One of the goals of our work is to facilitate parallel programming. Unfortunately, measuring productivity directly is not easy. Instead, we have measured the number of lines of code of the HTA and MPI programs and use this figure as an indirect measurement of productivity. Clearly, the number of lines of code is not the best metric of ease of programming, but in our case, it provides a reasonable estimate of the relative complexity of the programs. The plot in Figure 9 shows the lines of code for HTA and MPI codes. Each bar shows the lines of code for the Computation, Communication and Data Decomposition sections of the codes. As Figure 9 shows, HTA programs require significantly fewer lines of code. HTA programs have less computation lines than the MPI programs due to the use of vector operations, overloaded MATLAB^{TM}functions (CG and FT) and recursion (MG). Vector operations were used in all the applications when possible. However, we found that the MATLAB^{TM}JIT (Just In Time) compiler did not work on loops that contained vectorized and non-vectorized portions of code. As a result, some inner-loops that could have been vectorized were left non-vectorized for performance reasons. The lines of code for communication are significantly lower in HTA programs. HTA programs only need assignment instructions to perform communication, while in MPI programs, in addition to the send and receive instructions, packing and unpacking data and checking boundary conditions in the communication are also needed. HTA programs also have significantly fewer data decomposition instructions. HTA are partitioned and distributed using the single HTA constructor, while MPI programs need to compute a number of values including the limits of data owned by each processor, neighbors of a given processor, active set of processors in a given step of the program.

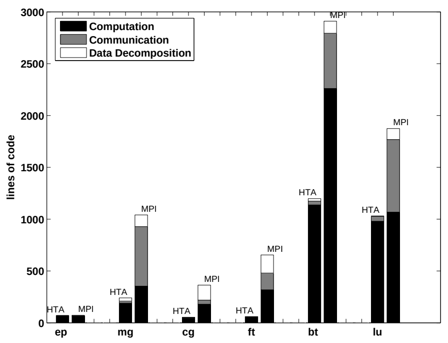


Figure 9: Linecount of key sections of HTA and MPI programs.

5 Related Work

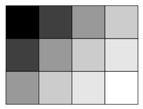
There has been significant research in parallel programming languages and libraries as mechanisms to efficiently utilize the computing power of distributed systems. Both programming languages and libraries can be classified by the view the programmer has of the computation. This sections discusses related work in two sections covering local and global views approaches.

5.1 Local View Approaches

In local view approaches, the programmer specifies the program execution path for each processor. SPMD programs which include calls to the standard message passing libraries such as MPI [11] or PVM [7] belong to this approach. This strategy usually delivers good performance, but at the expense of high programming costs because of the need to manually distribute data and synchronize processes. The SPMD model may lead to unstructured code where explicit communication is mixed with computations in the program. These features may make these programs difficult to write, understand and maintain. As a result, message-passing programming (or, more specifically, MPI programming) has been called the assembly language of the parallel programming.

A similar approach is followed by some programming languages such as Co-Array FORTRAN (CAF) [18], UPC [10] and Titanium [22]. These languages ease the development and readability of SPMD programs basically by replacing the calls to the message passing libraries with array declarations and assignments under the distributed shared memory model they provide. For instance, CAF adds an additional set of subscripts within square brackets to provide a straightforward representation of any access to non local data. CAF and UPC follow the SPMD model where data distribution and synchronization require much involvement from the programmer. For instance, in CAF each replication of the program is called an image. CAF programmers must determine the actual path of the program with the help of a unique image index by using normal sequential control constructs, which, as in the case of MPI, may lead to unstructured code.

```
for i=2:m-1
  for j=2:n-1
    a(i,j)= a(i-1,j) + ...
            a(i,j-1);
  end
end
```



$$I = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} \quad J = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

```
for K=2:M+N
  for i=2:dimx-1
     for j=2:dimy-1
        a{I+J == K}(i, j) = a{I+J == K}(i-1, j) + ...
                            a{I+J == K}(i, j-1);
     end
  end
  a{I+J == K+1 & I>1}(1, :) = a{I+J == K & I<M}(dimx-1,:);
  a{I+J == K+1 & J>1}(:, 1) = a{I+J == K & J<N}(:, dimy-1);
end
```

(a) (b) (c)

Figure 8: 2-D wavefront computation.(a) Fortran code. (b) Pictorial view. (c) HTA code.

5.2 Global View Approaches

In global view approaches programmers specify the global behavior of the algorithm from which the local, or per-processor, behavior is automatically obtained. Data are also global, and are dealt in a unified way. ¿From the programmer point of view, global view approaches are desirable, since they usually produce simpler code, what facilitates development and maintenance. The main concern with global view approaches is whether the resulting code performs well.

Global view languages include sequential language extended with directives for array distribution and alignment, loop scheduling and other details relevant to parallel computation. This is the approach followed by High Performance Fortran (HPF) [12, 15], where directives are used to annotate Fortran codes. One of the drawbacks of HPF is the gap between the goals programmers want to reach and the translation of the HPF compilers, aggravated by the fact that HPF directives are optional, so the compiler is free to ignore them. As a result, programmers may have difficulties controlling the execution of their HPF programs because it is not always easy to know the form of the target SPMD code. The lack of a clear performance model makes it difficult for programmers to reason about an algorithm's performance without a detailed knowledge of the compiler, and leads to unpredictable performance [17].

ZPL [8] is an example of a global view parallel language. A difference with HPF, is that ZPL has an explicit performance model that indicates where interprocessor communication takes place, as well as the type of communication that is needed. A characteristic of ZPL is the use of regions to refer to a collection of array indices. Region and region operators give programmers a mechanism for referring and operating on set of array elements, removing the need for explicit array indexing. The main disadvantage of ZPL is that it requires programmers to rewrite from scratch their applications to take advantage of this approach.

Another way to express parallelism is the approach followed in the project described above which uses classes that encapsulate the parallelism. These classes take care of the distribution of the data stored in their objects as well as the parallel computation, hiding the details from the programmer. STAPL [9] is another example of the use of classes to encapsulate parallelism The main differences with the approach discussed above is that STAPL focuses on data structures for symbolic computation while the focus of the HTA class is mainly numerical computations.

6 Conclusions

Most programmers today use low-level message-passing SPMD programming to implement their applications in distributed-memory systems because of its ability to attain high performance and because it allows them to reuse existing libraries and applications, since it is based on well-established sequential languages. Unfortunately, following the SPMD approach may lead to unstructured code, difficult to write and maintain.

We believe that the paradigm to program parallel computers should be based on a single-threaded global view of the application and its data. For this approach to succeed three issues must be addressed. First, parallel programming constructs should be compatible with sequential constructs. Reuse of existing applications and libraries is essential to users. Second, high-level approaches will only be widely adopted when their performance is competitive with that of the low-level message-passing programming. Since current compiler technology has many limitations, this means these approaches must be designed to give programmers control of execution and facilitate the task of compilers as much as possible. Finally, just as in the sequential languages, programmers should be able to statically determine, at least qualitatively, the performance behavior of their codes.

This paper reported on one approach that attempts to address these three issues. We argue that 1) the single thread execution model of the HTA not only releases programmers from the potential difficulties of SPMD parallel programming, but also provides high quality codes which are easy to develop and maintain. 2) HTA contains explicit information about tiling which could be used by a compiler to partition loops for locality or parallelism. 3) HTA encapsulates the parallelism and unifies the programming styles for single processor and multiprocessors. 4) HTA gives programmers clear information on when communication takes place and its cost based on its syntax and function calls, which are the basis for a well-defined performance model.

References

[1] Nas Parallel Benchmarks. Website. http://www.nas.nasa.gov/Software/NPB/.

[2] High Performance Fortran Forum. *High Performance Fortran Specification Version 2.0*, January 1997.

[3] G. Almási and D. Padua. MaJIC: Compiling MATLAB for Speed and Responsiveness. In *PLDI '02: Proceedings of the*

ACM SIGPLAN 2002 Conference on Programming language design and implementation, pages 294–303, New York, NY, USA, 2002. ACM Press.

[4] G. Almási, L. De Rose, B. B. Fraguela, J. Moreira, and D. Padua. Programming for Locality and Parallelism with Hierarchically Tiled Arrays. In *Proc. of LCPC 2003*, volume 2958 of *LCNS*, pages 162–176, College Station, Texas, Oct 2003. Springer-Verlag.

[5] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[6] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2), 2005. special issue on ”Program Generation, Optimization, and Adaptation”.

[7] A. Geist et al. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[8] B.L. Chamberlain et al. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.

[9] P. An et al. STAPL: An Adaptive, Generic Parallel Programming Library for C++. In *Proc. of LCPC*, pages 193–208, August 2001.

[10] W. Carlson et al. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): Portable Parallel Programming with the Message-Passing Interface”*. MIT Press, 1999.

[12] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-memory Machines. *Commun. ACM*, 35(8):66–80, 1992.

[13] P. Husbands and C. Isbell. Matlab*p: A Tool for Interactive Supercomputing. In *Procs. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[14] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley.

[15] C. Koelbel and P. Mehrotra. An Overview of High Performance Fortran. *SIGPLAN Fortran Forum*, 11(4):9–16, 1992.

[16] Y. Muraoka. Parallelism Exposure and Exploitation in Programs. Technical Report Report No. 71-424, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971.

[17] Ton Anh Ngo. *The role of performance models in parallel programming and languages*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1997.

[18] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[19] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[20] A. E. Trefethen, V. S. Menon, C. Chang, G. Czajkowski, C. Myers, and L. N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. Technical Report TR96-1586, May 1996.

[21] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44. ACM Press, 1991.

[22] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Workshop on Java for High-Performance Network Computing*, February 1998.