The Hierarchically Tiled Arrays Programming Approach *

Basilio B. Fraguela†, Jia Guo, Ganesh Bikshandi, María J. Garzarán,
Gheorghe Almási‡, José Moreira‡, and David Padua

Dept. of Computer Science
U. of Illinois at Urbana-Champaign
{jiaguo,bikshand,garzaran,
padua}@cs.uiuc.edu

†Dept. de Electrónica e Sistemas
Universidade da Coruña
Spain
basilio@udc.es

‡IBM Thomas J. Watson
Research Center
Yorktown Heights, NY, USA
{gheorghe,jmoreira}@us.ibm.com

ABSTRACT

In this paper, we show our initial experience with a class of objects, called Hierarchically Tiled Arrays (HTAs), that encapsulate parallelism. HTAs allow the construction of single-threaded parallel programs where a master process distributes tasks to be executed by a collection of servers holding the components (tiles) of the HTAs. The tiled and recursive nature of HTAs facilitates the adaptation of the programs that use them to varying machine configurations, and eases the mapping of data and tasks to parallel computers with a hierarchical organization. We have implemented HTAs as a MATLAB^{TM} toolbox, overloading conventional operators and array functions such that HTA operations appear to the programmer as extensions of MATLAB^{TM}. Our experiments show that the resulting environment is ideal for the prototyping of parallel algorithms and greatly improves the ease of development of parallel programs while providing reasonable performance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Parallel languages

1. INTRODUCTION

Parallel programs are difficult to develop, maintain and debug. This is particularly true in the case of distributed memory machines, where data exchanges involve message passing. Moreover, performance in parallel programs depends on many more factors than in sequential ones. Systems may be heterogeneous; the architecture to consider involves a network for the communications and different layers of operating system and user libraries may be involved in the passing of messages. As a result, performance tuning is also much harder. The language and compiler community have come up with several approaches to help programmers deal with these issues.

One of the approaches to simplify the development of distributed memory programs is to use standard message passing libraries like MPI [9] or PVM [8] which enable the portability of the parallel applications. Still, data distribution and synchronization must be completely managed by the programmer. Also, the SPMD programming model of the codes that use these libraries creating room for unstructured codes in which, for example, communication may take place between widely separated sections of code and in which a given communication statement could interact with several different statements during the execution of the program. Some programming languages like Co-Array FORTRAN [12] and UPC [5] improve the readability of the programs by replacing explicit communications with array assignments, but they still have all the drawbacks of the SPMD approach.

Another strategy to improve the programmability of the distributed memory environments consists in using a single thread of execution and letting the compiler take care of the distribution of the data and the schedule of parallel tasks. This is for example the approach of the High Performance Fortran[10, 11]. Unfortunately, compiler technology does not seem to have reached a level in which compilers following this approaches can generate competitive code.

In this paper we explore the possibility of extending a single-threaded object-oriented programming language with a *new class, called Hierarchically Tiled Array or HTA [3], that encapsulates the parallelism in the code.* The objects of this class are tiled arrays whose tiles can themselves be recursively tiled. The tiles and the operations on them are distributed among a collection of servers. The HTA class provides a flexible indexing scheme for its tiles and allows data communication between servers to be expressed by

*This work has been supported in part by the Defense Advanced Research Project Agency under contract NBCH30390004. This work is not necessarily representative of the positions or policies of the U.S. Army or Government. It has also been supported in part by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.

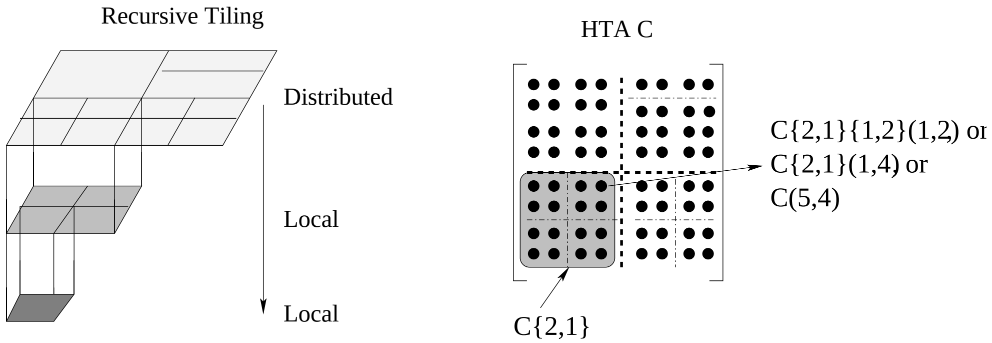

Figure 1: Pictorial view of a hierarchically tiled array.

means of array assignments and operations. Many of these operations that represent array communication overload standard operators such as circular shift and transpose. As a result, HTA based programs look as if they had a single thread of execution, but are actually executed on a number of processors. This improves the readability and ease of development and maintenance of HTA code. Furthermore, the compiler support required by our parallel programming approach is minimal, since the implementation of the class takes care of the parallelization. Also, thanks to the recursive nature of the tiles in the HTAs, data mapping and tasks scheduling in computers with a hierarchical organization can be done very naturally. As a proof of concept we have implemented the HTA class in MATLAB$^{\mathrm{TM}}$, which contains a high-level programming language with object-oriented features that is easy to extend.

The rest of this paper is structured as follows. HTA syntax and semantics are briefly described in the next Section. Section 3 provides several code examples. Our implementation of the HTA class is described in Section 4. Then, Section 5 evaluates the performance of this implementation and compares its ease of use with that of a traditional MPI+C/Fortran approach. In Section 6 an analytical comparison of HTA with other related languages is given. The last section is devoted to our conclusions and future work.

2. HIERARCHICALLY TILED ARRAYS

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. These definitions do not require all tiles to have the same size, or that tiles have the same internal partitioning. HTAs can be used to facilitate the expression of both locality and parallelism. The idea is to distribute across processors the outermost tiles of a HTA for parallelism, and use the inner tiles for locality. In the case of sequential programs all the tiles will be used for locality. Fig. 1 shows an example of a legal HTA with two levels of tiling.

2.1 Accessing the Contents

References to HTAs allow access to both tiles and elements. Curly brackets are used when indexing tiles, while parenthesis denote the access to elements within the HTA or its tiles. Fig. 1 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 a matrix. The same element can also be accessed by selecting the bottom-level tile that contains it and its relative position inside this tile. The expression `C{2,1}{2,1}(1,4)` refers to the same datum `C(5,4)`. A third possibility to reference `C(5,4)` is by selecting the top-level tile that contains the element and flattening or disregarding its internal tiled structure: `C{2,1}(1,4)`.

In any kind of indexing, a range of element position or tiles may be chosen in each dimension using triplets of the form *begin:step:end*, where the step is optional. If no step is provided, a step one is assumed. Also, the `:` notation can be used in any index to refer to the whole range of possible values for that index. This way, for example, `C(:,1:3)` uses flattening to refer to the first three columns of elements of the HTA C; and `C{2,:}(1:2:4,1:3)` refers to the first three elements of the odd rows of the two lower top-level tiles of C.

2.2 Binary 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). 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 tiles of the HTA, provided that the tiles and the array are conformable; while in the case of the scalar, the operation takes place at the element level. Again, the output HTA has the same topology as the input HTA.

2.3 Assignments

The semantics for assignments to HTAs have similarities with those for binary operators. When a scalar is assigned to a range of positions within an HTA, the scalar gets replicated in all of them. When an array is assigned to a range of tiles of an HTA, the array is replicated in all of the tiles, provided such an assignment is legal. Finally, an HTA can be assigned to another HTA (or a range of tiles of it) if the copy of the correspondingly selected tiles from the right-hand side (RHS) HTA to those selected in the left-hand side (LHS) HTA is legal. The condition for

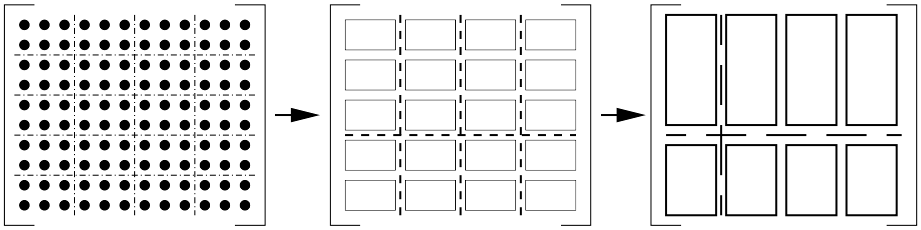

Figure 2: Bottom up tiling.

an assignment to be legal is that once it takes place, the adjacent tiles in the resulting HTA continue to have the same size along each dimension of adjacency, that is, the resulting HTA continues to fulfill the properties of an HTA.

2.4 Execution Model

The machine model for an HTA program is that of a client, which runs the main thread of execution, and that is connected to a distributed memory machine with an array of processors, called servers, onto which the top-level tiles of the HTAs are mapped. Whenever an operation found in the code that the client executes involves the distributed tiles of an HTA, such operation is broadcasted from the client to the servers so that they execute it in parallel. When the operation only involves tiles that the server owns, the server performs locally the computation. If, however, the computation requires tiles that the server does not own, it first requests them to the owner servers, and then it performs the computation. Thus, in a program using HTAs the parallelism and the communication is encapsulated in the statements that operate on tiles of one or more HTAs.

While this is the execution model from the point of view of the programmer, and it is the way our current implementation works, HTA programs could also be translated by a compiler into tasks that execute in the nodes of the array of processors synchronizing and exchanging data when required. This is perfectly feasible approach that would improve the scalability of this programming approach.

2.5 Construction of HTAs

The simplest way to build an HTA is by providing a source array and a series of delimiters in each dimension where the array should be cut into tiles. For example, if M is a 1000×1000 matrix, an HTA resulting from its partitioning in tiles of 100×250 elements would be created by the statement:

```
A = hta(M, {1:100:1000,1:250:1000});
```

The triplet with the curly brackets are the *partition vector* for each dimension of the source array. The elements in each partition vector specify the hyperplanes that cut the input matrix along the corresponding dimension to distribute it in tiles. The elements in the partition vector mark the beginning of each sub-tile. This constructor can also be used to create HTAs with different levels of tiling using a bottom-up approach. For example, given a 10×12 matrix D, the statements

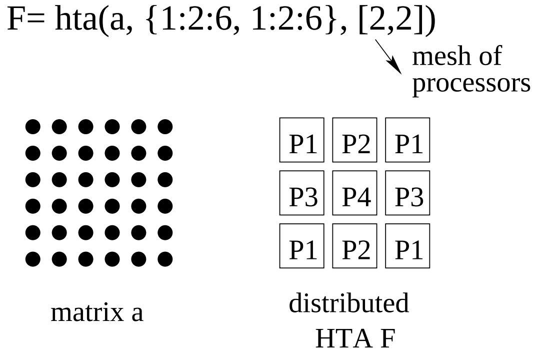

Figure 3: Mapping of tiles to processors.

```
C = hta(D, {[1,3,5,7,9],[1,4,7,10]});
B = hta(C, {[1,4],[1,2,3,4]});
A = hta(B, {[1,2],[1,2]});
```

will generate the three HTAs shown in Fig. 2. Notice that using this bottom-up approach matrix A can also be created using a single statement

```
A = hta(D, {[1,3,5,7,9],[1,4,7,10]}, ...
           {[1,4],[1,2,3,4]}, ...
           {[1,2],[1,2]});
```

where the ... just mean the continuation of the command in the following line.

Finally, it is also possible to build empty HTAs whose tiles are later filled in. To build one, the HTA constructor must be called with the number of desired tiles per dimension. For example, F = hta(3, 3) would generate an empty 3×3 HTA F.

The examples discussed above generate non-distributed HTAs, which are located only in the client. Nevertheless, most of the times we will be interested in generating HTAs whose contents are distributed on a mesh of processors, so that we can operate in parallel on its tiles. Our toolbox currently supports a single form of distribution. Namely, it can distribute the top level tiles of an HTA cyclically on a mesh of processors. This corresponds to a block cyclic distribution of the matrix contained in the HTA, with the blocks defined by the top level partition. In order to achieve this, the constructor of the HTA needs a last parameter that specifies the dimensions of the mesh by means of a vector. Fig. 3 shows an example where a

```
a = hta(MX,{dist}, [P 1]);
b = hta(P, 1, [P 1]);
b{:} = V;
r = a * b;
```

Figure 4: Sparse matrix vector product.

```
for i = 1:n
  c = c + a * b;
  a = circshift(a, [0, -1]);
  b = circshift(b, [-1, 0]);
end
```

Figure 5: Main loop in Cannon's algorithm.

6×6 matrix is distributed on a 2×2 mesh of processors as the the last parameter of the HTA constructor indicates. In the future we plan to offer more mappings and a representation for hierarchical organizations of processors.

3. PARALLEL PROGRAMMING USING HTAS

In this section we illustrate the use of HTAs with five simple code examples.

3.1 Sparse Matrix-Vector Product

Our first example, sparse matrix-vector product (Fig. 4), illustrates the expressivity and simplicity of our parallel programming approach. This code multiplies a sparse matrix MX by a dense vector V using P processors. We begin by distributing the contents of the sparse matrix MX in chunks of rows into an HTA a by calling an HTA constructor. The P servers handling the HTA are organized into a single column. We rely on the dist argument to distribute the array MX in such a way that it results in a uniform computational load across the servers.

Next we create and empty HTA b distributed across all processors. We assign the vector V to each of the tiles in hta b (b{:}=V). With this assignment, the vector V is copied to each tile of the hta b. Since HTA b is distributed across the P processors, this copy requires that the client broadcasts V to all the servers that have a tile of the HTA b. Notice that since HTA b and a have the same number of tiles and they are mapped to the same processor mesh, each processor holding a tile of a, will now hold a copy of V too.

The multiplication itself is in the last line of the code, where the binary operator * is invoked on a and b. The effect is that corresponding tiles of a and b, which are located in the same server, are multiplied, giving place to a distributed matrix-vector multiply. The result is a HTA r, distributed across the servers with the same mapping as the inputs. This HTA can be flattened back into a vector containing the result of the multiplication of a by b by using the r(:) notation.

The code completely hides the fact that MX is sparse because MATLAB^{TM} provides the very same syntax for dense and sparse computations, a feature our HTA class implementation in MATLAB^{TM} has inherited.

3.2 Cannon's Algorithm for Matrix Multiplication

While the previous example only required communication between the client, which executes the main thread, and each individual server, Cannon's matrix multiplication algorithm [4] is an example of code that also requires communication between the servers.

The algorithm has $O(n)$ time complexity and uses $O(n^2)$ processors (servers). In our implementation of the algorithm, the operands, denoted a and b respectively, are HTAs tiled in two dimensions which are mapped onto a mesh of $n \times n$ processors.

In each iteration of the algorithm's main loop each server executes a matrix multiplication of the tile of a and b that currently reside on that server. The result of the multiplication is accumulated in a (local) tile of the result HTA, c. After the computation, the tiles of a and b are circular-shifted as follows: the tiles of b are shifted along the first dimension; the tiles of a are shifted along the second dimension. The effect of this operation is that the tiles of a are sent to the left processor in the mesh and the tiles of b are sent to the upper processor in the mesh. The left-most processor transfers its tile of a to the right-most processor in its row and the bottom-most processor transfers its tile of b to the top-most processor in its column.

At the end of n iterations each server holds the correct value for its associated tile in the output HTA c=a*b. Fig. 5 shows the main loop of Cannon's algorithm using HTAs.

3.3 Jacobi Relaxation

Referencing arbitrary elements of HTAs results in complex communication patterns. The blocked Jacobi relaxation code in Fig. 6 computes the new value of each element as the average of its four neighbors. Each block of d×d elements of the input matrix is represented by a tile of the HTA v. In addition, the tiles also contain extra rows and columns for use as border regions for exchanging information with the neighbors. As a result, each tile has d+2 rows and d+2 columns. This situation is depicted in Fig. 7, which shows an example 3×3 HTA v with d= 3. The inner part of each tile, surrounded by a dotted line in the picture, contains the data from the input matrix. The first and last row and column in each tile are the shadows that will receive data from the internal tile of the neighbors in order to calculate locally the average of the four neighbors for each element. This exchange of shadows is executed in the first four statements of the main loop and it is also illustrated in Fig. 7, which shows the execution of the statement v{2:n,:}(1,:) = v{1:n-1,:}(d+1,:). As we see, by using HTA addressing, communication and computation can be easily identified by looking at the tiled indexes.

In this case the flattened version of the HTA v does not quite represent the desired end result, because of the existence of the border exchange regions. However, the desired matrix can be obtained by first removing the border regions and applying the flattening operator afterward: (v{:,:}(2:d+1,2:d+1))(:,:).

3.4 Embarrassingly Parallel Programs

Easy programming of embarrassingly parallel and MIMD style codes is also possible using HTAs thanks to the parHTAFunc

```
while ~converged
  v{2:n,:}(1,:) = v{1:n-1,:}(d+1,:);
  v{1:n-1,:}(d+2,:) = v{2:n,:}(2,:);
  v{:,2:n}(:,1) = v{:,1:n-1}(:,d+1);
  v{:,1:n-1}(:,d+2) = v{:,2:n}(:,2);

  u{:,:}(2:d+1,2:d+1) = K * (v{:,:}(2:d+1,1:d) + v{:,:}(1:d,2:d+1) + ...
                             v{:,:}(2:d+1,3:d+2) + v{:,:}(3:d+2,2:d+1));
end
```

Figure 6: Parallel Jacobi relaxation

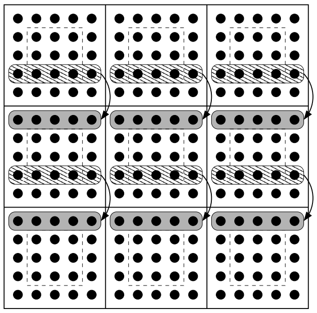

Figure 7: HTA `v` in the Jacobi relaxation code. The shared areas indicate the regions of data that are moved in the first statement of the loop.

```
input = hta(P, 1, [P 1]);
input{:} = eP;
output = parHTAFunc(@estimatePi, input);
myPi = mean(output(:));

function r = estimatePi(n)
  x = randx(1, n);
  y = randx(1, n);
  pos = x .* x + y .* y;
  r = sum(pos < 1) / n * 4;
```

Figure 8: Estimation of π using parallel Monte Carlo

function. It allows the execution of a function in parallel on different tiles of the same HTA. A call to this function has the form `parHTAFunc(func, arg1, arg2, ...)`, where `func` is a pointer to the function to execute in parallel, and `arg1`, `arg2`,... are the arguments for its execution. At least one of these arguments must be a distributed HTA.

The function `func` will be executed in the servers that hold the tiles of the distributed HTA. For each local execution, the distributed HTA in the list of input arguments is replaced by the local tile of each server. If the server keeps several tiles, the function will be executed for each of these.

Several arguments to `parHTAFunc` can be distributed HTAs. In this case they all must have the same number of tiles in every dimension and the same mapping. This way, in each local execution, the corresponding tiles of the HTAs, which reside in the same server, are used as inputs for `func`. The result of the parallel execution is stored in an HTA (or several, if the function has several outputs) that has the same number of tiles and distribution as the input distributed HTA(s).

Fig. 8 illustrates a `parHTAFunc` that estimates π using the Monte Carlo method on `P` processors. A distributed HTA `input` with one tile per processor is built. Then its tiles are filled with `eP`, the number of experiments to run in every processor. The experiments are made on each processor by the function `estimatePi` defined in Fig. 8, whose input argument is the number of experiments to make. MATLAB^{TM} syntax is used throughout the code to define the `estimatePi` function, designate its pointer `@estimatePi`, and the different operations required by the function, such as `.*`, the element-by-element product of two arrays. The function `randx` is similar to `rand` in MATLAB^{TM} but generates a different sequence on each processor. The result of the parallel execution of the function is a distributed HTA `output` that has the same mapping as `input` and keeps a single tile per processor with the local estimation of π. To compute the global estimation, which is the mean of these values, the standard MATLAB^{TM} function `mean` is applied to the flattened `output`.

The global estimation is the mean of these values, so the standard MATLAB^{TM} function `mean` is applied to the flattened `output` to calculate this value.

3.5 Non-Numerical Problems

All the previous examples are numerical computing. We often wonder whether new parallel programming paradigms are only suitable for this kind of applications. For this reason, in this section we describe a parallel HTA implementation of quicksort. The code is shown in Fig. 9. The program sorts an input vector `v` with `num_el` elements using `P` processors in $\log_2(P)$ steps.

The program uses the Hyperquicksort algorithm [13]. The algorithm regards the mesh of processors as a linear array where the processors are numbered from 1 to `P`. The algorithm starts by distributing the input vector `v` among the processors. Then, each processor sorts each subset using the sort MATLAB^{TM}function. The algorithm proceeds in $\log_2(P)$ iterations starting with i ranging from $\log_2(P)$ to 1. Each iteration i divides the processors into sets of 2^i consecutive processors. Each processor set has a pivot that is used to partition the data. At each iteration i, the data is rearranged such that the 2^{i-1} processors in the upper half of the processor set contain the values greater than the pivot, and the processors in the lower half contain the smaller values.

Each iteration starts by choosing a pivot in each tile of the HTA `h` in which the input vector has been partitioned. This is done by applying the function `pivotelement` in parallel in every chunk, which chooses the value in the middle of the chunk, or 0 if the tile is empty. Then, the

```
num_el = length(v);
h = hta(v, {1:ceil(num_el/P):num_el}, [P 1]);
h = parHTAFunc(@sort, h);
for i = log2(P):-1:1

  % First stage
  pivot = parHTAFunc(@pivotelement, h);
  initSet = 1:2^i:P;
  pivot{:} = pivot{sort(repmat(initSet, 1, 2^i))};

  % Second stage
  [l, u] = parHTAFunc(@partition, h, pivot);

  % Third stage
  tmpU = initSet;
  for j=1:2^(i-1)-1
    tmpU = [tmpU initSet+j];
  end
  tmpU = sort(tmpU);
  tmpL = setdiff(1:P, tmpU);

  tmptu = u{tmpU};
  u{tmpU} = l{tmpL};
  l{tmpL} = tmptu;

  % Fourth stage
  h = parHTAFunc(@cat_sort, l, u);
end

% The following functions process each tile
function t = pivotelement(h)
  n = size(h,1);
  half = ceil(n/2);
  if half
    t = h(half);
  else
    t = 0;
  end

function [l, u] = partition(h, pivot)
  l = h(h<=pivot);
  u = h(h>pivot);

function r = cat_sort(l, u)
  r = sort(cat(1, l, u));
```

Figure 9: Parallel quicksort

pivot in the first processor in each set (variable `initSet` keeps the indexes of these processors) is broadcasted to the other processors in the set by means of an assignment in the HTA `pivot`.

In the second stage of the algorithm, each processor partitions its local chunk in two subsets by running the function `partition`. For each processor, the tiles of the HTA `l` contain the elements that are smaller than or equal to the pivot; while the tiles of the HTA `u` contains the greater ones.

In the third stage, the tiles of HTAs `l` and `u` are exchanged within each half of each set of 2^i consecutive processors. The processors in the first half of each set send their `u` tiles to the processors in the second half, which, in return, will send their `l` tiles. The swapping of tiles is achieved by means of assignments, after calculating the correct indexes for the operation. Notice that once the exchange completes, all the data in the tiles of both `l` and `u` in the first 2^{i-1} processors of each set is smaller than (or equal to) the pivot used for the sorting of the set; while the data in the tiles of the processors of the second half of the set is bigger than the pivot.

The fourth and final stage of the algorithm takes care of fusing the tile of `l` and the tile of `u` that reside in each processor and sorting locally the result. This is achieved by applying in parallel the function `cat_sort` to the HTAs `l` and `u`. The result is a regenerated HTA `h`.

4. IMPLEMENTATION

HTAs can be added to almost any object-based or object-oriented language. We chose the MATLAB^{TM} environment as the host for our implementation for a number of reasons:

- MATLAB^{TM} is a linear algebra language with a large base of users who write scientific code. HTAs allow these users to harness the power of a cluster of workstations instead of a single machine.
- MATLAB^{TM} is polymorphic, allowing HTAs to substitute regular arrays almost without changing the rest of the code, thereby adding parallelism painlessly.
- MATLAB^{TM} is designed to be extensible. Third party developers can provide so-called *toolboxes* of functions for specialized purposes. MATLAB^{TM} also provides a native method interface called MEX, which allows functions to be implemented in languages like C and Fortran.

In the MATLAB^{TM} environment, HTAs are implemented as a new type of object and they are built through the constructors described in Section 2.5. The bulk of the HTA implementation is actually written in C and interfaces with MATLAB^{TM} through MEX. In general, methods that do not involve communications, such as those that test the legality of operations, were written in MATLAB^{TM} to simplify their development. Small methods used very frequently were written in C for performance reasons. Both client and servers have a copy of matlab and HTA system on top of it. Communications between the client and the servers are implemented using the MPI [9] library, thus methods that involve communication were written in C in order to use the message-passing library.

Our implementation supports both dense and sparse matrices with double precision data, which can be real or complex. Any number of levels of tiling is allowed in the HTAs, although every tile must have the same number of levels of decomposition in order to simplify the legality tests. In practice this is not an important restriction, since an HTA can consist of a single tile that holds another HTA or a matrix. Also, the current implementation requires the HTAs to be either full (every tile has some content) or empty (every tile is empty).

4.1 Internal Structure

The architecture of our MATLAB^{TM} implementation is shown in Fig. 10. MATLAB^{TM} is used both in the client, where the code is executed following a single thread, and in the servers, where it is used as a computational engine for the distributed operations on the HTAs. All the

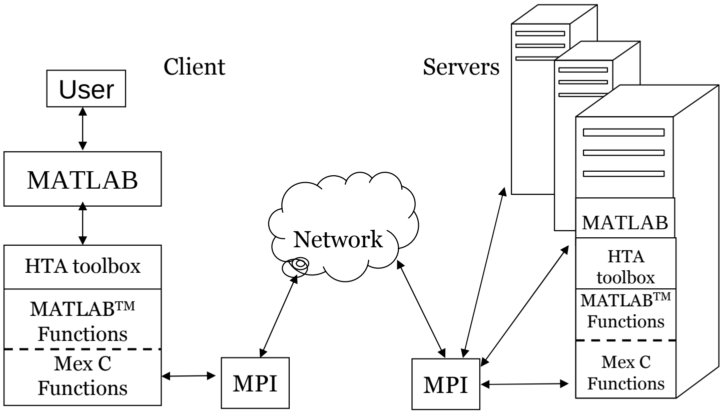

Figure 10: HTA implementation in MATLAB$^{\text{TM}}$

communications are done through MPI; the lower layers of the HTA toolbox take care of the communication requirements, while the higher layers implement the syntax expected by MATLAB$^{\text{TM}}$ users.

HTA programs have a single thread that is interpreted and executed by the MATLAB$^{\text{TM}}$'s client. The HTAs are just objects within the environment of the interpreter, that when referenced generate calls to the methods of their class. When an HTA is local, that is, when it is not distributed on the array of servers, the client HTA keeps both the structure and the content of the HTA. When it is distributed, the HTA in the client holds the structure of the HTA at all its levels, although it does not contain its content. It also keeps the information of the mapping of the top level tiles of the HTA on the mesh of servers. In this way, for any operation, regardless whether the HTA is local or distributed, the client is able to locally:

- test the legality of the operation
- calculate the structure and mapping of the output HTA(s)
- send the messages that encode the command and its arguments to the servers.

Usually the legality check involves no communication, and the client is able to generate locally the output HTAs of the distributed operations. An advantage of this approach is once the client broadcasts the commands, it is free to continue and execute the next statement/operation. On the other hand, doing all checking in the client creates a bottleneck in the client, thus serializing part of the execution.

A source of inefficiency in our current implementation is that the client uses point to point communication rather than broadcasting. This was done so that servers would know that every message a server receives corresponds to an operation in which it participates. This saves the tests that would be required if the messages were broadcasted. This strategy reduced the development time for the toolbox, but it is slower than using the broadcast, particularly as the number of servers increases.

The servers have the structure and data of the top level HTA tiles that are mapped to them, but they also know the structure of the HTAs that are not mapped to them. So, when a HTA is not mapped to a server, the server knows the number of dimensions of the HTA, the number of tiles per dimension at the top level, and the size of each top level tile in each dimension in terms of elements, as well as which servers owns each tile. In practice, this allows the servers very often to calculate which server(s) must be the destination or the source of their messages when they need to exchange data.

4.2 MATLAB™Specific Implementation Issues

A difficulty we found is the lack of destructor in the MATLAB$^{\text{TM}}$ classes. This hindered our ability to destroy distributed HTAs because the HTA subsystem is not notified when an HTA instance is destroyed by the MATLAB$^{\text{TM}}$.

Our solution to this problem is to apply a garbage collection mechanism based on a pattern of changes that we have detected in the internal structure of an HTA when MATLAB$^{\text{TM}}$ destroys it. Whenever a distributed HTA is built, a record of its internal structure is created in the client. The constructor checks for changes in other distributed HTA structures. If any changes are detected, we know that the corresponding HTA has been destroyed and we ask the servers to destroy their components of the deleted HTAs.

Our implementation follows the copy-on-write behavior of MATLAB$^{\text{TM}}$. In a MATLAB$^{\text{TM}}$ assignment, the RHS is not actually copied, but rather a reference is created from the LHS of the assignment to the old value. A copy is only made when the user attempts to modify the data referred to. Since our HTAs are implemented with the same variables that MATLAB$^{\text{TM}}$ uses, an HTA can have multiple references to it. Private copies of distributed HTAs are made only when they have several handles/names in the program and some of them modify the contents of the HTA.

5. EVALUATION

5.1 Environmental Setup

Our toolbox has been implemented and tested in an IBM SP system, an HP HPC320 server with alpha processors and True64 OS, and in a small cluster of PCs with Linux. We present experimental results for the IBM SP system. Most of our measurements were done on in IBM SP system because of its higher availability, larger number of processors and more MATLAB^{TM} licenses available to us.

Our IBM SP system consists of two nodes. Each node has 8 Power3 processors that run at 375 MHz and 8 GB of memory. Two configurations have been used in the experiments: one with 2×2 mesh of 4 servers, and another one with 3×3 mesh of 9 servers. Each node contributes half of the servers. There is an additional processor that executes the main thread of the program, thus acting as the client or master of the system.

The computational engine and interpreter for our HTA toolbox was implemented using MATLAB^{TM} R13 (Section 4). The C files of the toolbox were compiled with the VisualAge C xlc compiler for AIX, version 5.0, with the `O3` flag. The MPI library used was the one provided by the IBM parallel Environment for AIX, version 3.1. Serial and parallel MATLAB^{TM}benchmarks were executed in this framework.

Two series of benchmarks were used to evaluate the HTA toolbox. The first one consists of five simple kernels: *smv* implements the sparse matrix vector product shown in Fig. 4 and described in Section 3.1; *cannon* is an implementation of Cannon's matrix multiplication algorithm [4], whose main loop is depicted in Fig. 5; *jacobi* is the Jacobi relaxation code shown in Fig. 6; *summa* represents the SUMMA [7] matrix multiplication algorithm; and finally *lu* is an implementation of the LU factorization algorithm. Our second set of benchmarks consists of the well-known *ep*, *mg*, *ft* and *cg* NAS benchmark, version 3.1 [1].

Four versions of each benchmark were written: two serial programs, one in MATLAB^{TM} and the other one in C/Fortran, and two parallel versions, one implemented using HTAs and the other one using MPI+C/Fortran. In the MPI implementation of the benchmarks the blocks were mapped to the mesh of processors in the same way the HTAs do.

The computational engine and interpreter for our HTA toolbox was implemented using MATLAB^{TM} R13 (Section 4). The C files of the toolbox were compiled with the VisualAge C xlc compiler for AIX, version 5.0, with the `O3` flag. The MPI library used was the one provided by the IBM parallel Environment for AIX, version 3.1. Parallel MATLAB^{TM}benchmarks were executed in this framework. Serial benchmarks were executed using the plain MATLAB^{TM} R13.

The serial C and MPI implementations for cannon, summa and lu were implemented using the IBM ESSL library, while the corresponding MATLAB^{TM}HTA codes use the MATLAB^{TM} libraries, which are less optimized. The C codes were compiled with the xlc compiler with the `O3` flag. The fortran codes (serial and MPI versions of NAS benchmarks) were compiled using the xlf compiler from IBM also with the `O3` flag. The MPI library used was MPI IBM Parallel Environment for AIX, version 3.1.

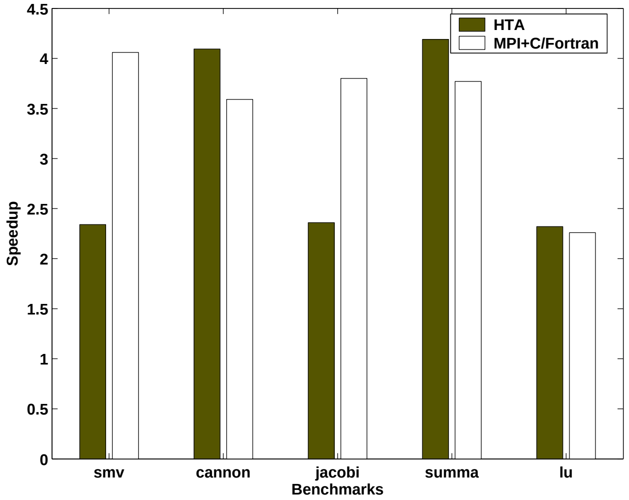

Figure 11: Speedup for HTA and MPI+C/Fortran on 4 servers for the simple kernels

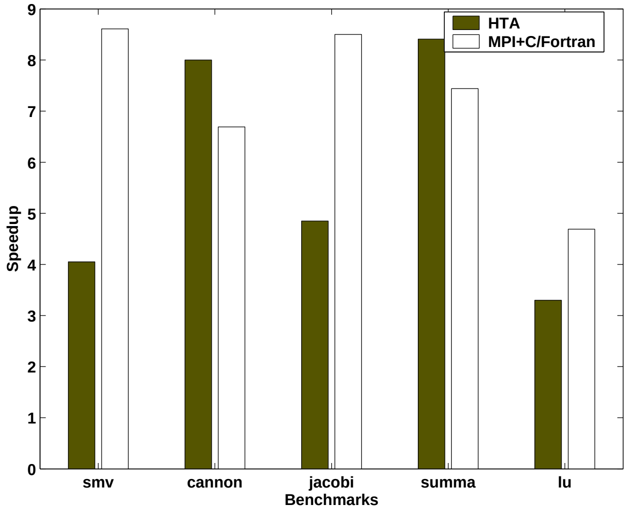

Figure 12: Speedup for HTA and MPI+C/Fortran on 9 servers for the simple kernels

Finally, notice that although the main focus of our work is the achievement of parallelism, HTAs can also be used to express locality. In fact, we have run experiments that show that when multiplying matrices of sizes larger than 2000×2000, an implementation based on HTAs, but executed serially, can run 8% faster than the native MATLAB^{TM} matrix product routine.

In the next Sections we evaluate the performance and the ease of programming of the MPI and the HTA approach to write parallel programs.

5.2 Performance Analysis

5.2.1 Simple benchmarks

Figs. 11 and 12 show the speedup obtained by the simple kernels for the HTA and MPI+C/Fortran when running on 4 and 9 servers, respectively. The speedup of the HTA parallel versions is computed from the parallel MATLAB^{TM}

HTA execution time and the corresponding execution time of the serial MATLAB^{TM} . Likewise, the speedup of the MPI approach is computed using the execution time of the sequential C/Fortran and the MPI+C/Fortran programs. For both configurations, 4 and 9 servers, all the benchmarks are evaluated using matrices of size 4800 × 4800. The input sparse matrix of *smv* has a density of 20%.

The figures show that the speedups of *cannon* and *summa* using HTAs are similar or even better than those of the MPI+C/Fortran programs. However, for smv, jacobi and lu the speedups of the MPI+C/Fortran versions is better than the speedups of the HTA ones, except the speedup of *lu* using 4 processors is similar in both cases.

The low speedups of the HTA codes are due to extra overheads built into our implementation. One cause of extra overhead is the broadcast of HTA commands to the servers: all HTA operations need to be broadcast from the client, who executes the main thread, to the servers, before they get executed. This approach simplifies implementation and implicitly synchronizes the processors, which in turn eases programming; but if the HTA operation handles relatively small amounts of raw data, broadcast overhead dominates execution time. By comparison, in an MPI program this broadcast is unnecessary because program execution is governed locally.

Another source of overhead is due to the limitations of the MATLAB^{TM} extension interface. Whenever a subset of positions of a tile is to be modified in an HTA method, a copy of the whole tile must be done. Copying an HTA in every assignment is another source of overhead. This effect is particularly visible in the speedups of *lu* and *jacobi* because they require the modification of portions of tiles rather than whole tiles. The reason for this copy is that in MATLAB^{TM} every parameter is passed to user-defined functions and methods by value. MATLAB^{TM} itself does not suffer from this overhead, because the indexed assignment operation is a built-in function to which arguments are passed by reference.

As we see, the current sources of overheads are mostly due to implementation issues and not inherent limitations of the HTA approach. The overhead due to the broadcast of the HTA commands can be mitigated by sending pre-compiled snippets of code to the servers for execution (although this implies the existence of a compiler). By re-implementing the indexed assignment method in C instead of MATLAB^{TM} we can mitigate the overhead caused by excessive copying of HTAs.

Finally, notice that the absolute execution time of the parallel MATLAB^{TM} programs based on HTAs is at most a factor of two slower than their MPI counterparts. This shows that the overhead introduced by MATLAB^{TM} in these codes is not very high and therefore the speedups obtained are mainly the result of parallelizing useful computation and not due to parallelization of the overhead.

5.2.2 NAS benchmarks

Figures 13 to 16 show the absolute execution time and the speedup obtained using 4 and 8 servers (processors) for both MPI and HTA programs for the NAS benchmarks, which must be run using a number of servers that is a power of two. The labels *ft_1d* and *ft_2d* correspond to *ft* with 1-D and 2-D decomposition respectively. That is the 3-D array whose forward and inverse FFT are calculated in the benchmark is partitioned either along only one (*ft_1d* or two *ft_2d* of its dimensions. We use the inputs of the class A of the NAS benchmarks. So, the size of the input for *mg* is a $256 \times 256 \times 256$ array, while for *ft* it is $256 \times 256 \times 128$. For the kernel *ep* 536870912 random numbers are generated; finally, the input for *cg* is a 14000 × 14000 sparse matrix with 1853104 nonzeros and a vector of size 14000.

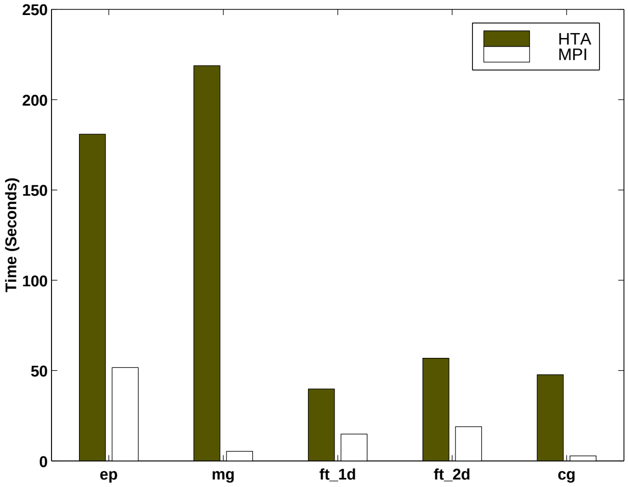

Figure 13: Execution time for HTA and MPI+C/Fortran on 4 servers for the NAS benchmarks

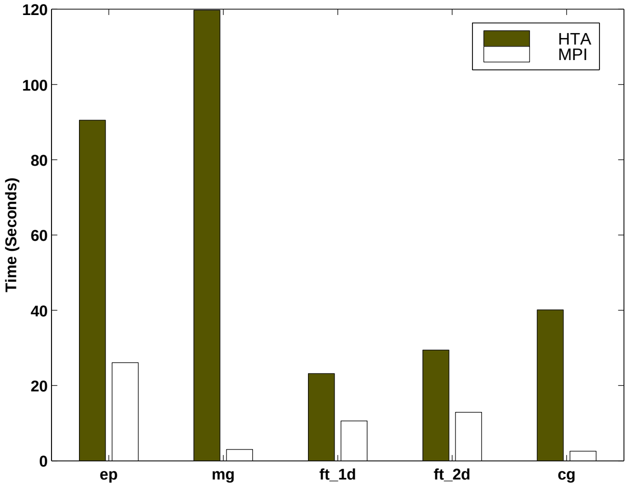

Figure 14: Execution time for HTA and MPI+C/Fortran on 8 servers for the NAS benchmarks

Figures 13 and 14 show that the parallel execution time of the HTA implementation takes longer than the MPI one. However, remember that our HTA system is built on top of MATLAB^{TM} which is a slow interpreted environment. However, Figures 15 and 16 show that the speedup of each benchmark is significant.

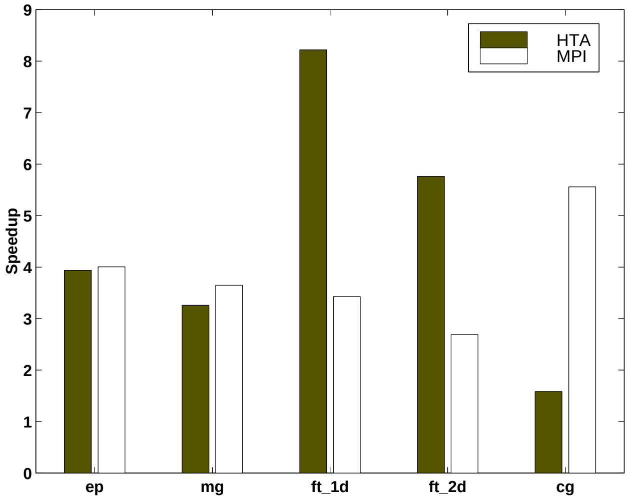

Figure 15: Speedup for HTA and MPI+C/Fortran on 4 servers for the NAS benchmarks

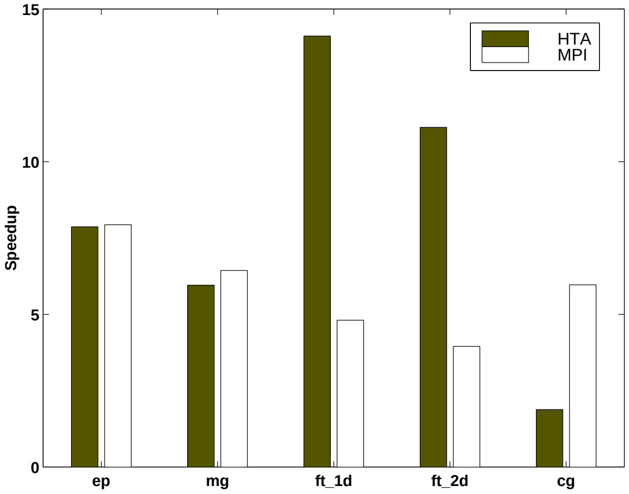

Figure 16: Speedup for HTA and MPI+C/Fortran on 8 servers for the NAS benchmarks

The embarrassing parallelism of the kernel *ep* allows it to achieve a perfect speedup, since there is no communication overhead apart from the initial distribution and final reduction. The kernel *ft* has a super-linear speedup for the HTA implementation. In *ft* the timing also includes the initialization of the complex array with random entries. This takes a long time in a serial MATLAB$^{\mathrm{TM}}$ program, which we believe is due to cache and/or TLB misses. The speedup of the HTA version of the kernel *mg* is slightly smaller than that of its MPI version. Finally, the kernel cg does much better in MPI than with HTAs; mainly due to the high optimization of the somewhat irregular patterns of communication in the MPI version. The reasons for the additional overheads discovered in these programs are the same as the ones commented before for the simple kernels. The readers should also note that the NAS benchmark kernels are highly optimized, while our current HTA versions are not that optimized.

5.3 Ease of programming

Table 1 shows the lines of code used by each benchmark with each of the 4 different implementations: MATLAB$^{\mathrm{TM}}$, HTA, C/Fortran, and MPI. Unsurprisingly, MATLAB$^{\mathrm{TM}}$ and HTA based programs take fewer lines of code than C, Fortran and MPI programs. In the case of *smv*, *cannon*, *jacobi*, *summa* and *lu* the difference is of an order of magnitude. For the NAS benchmarks, the HTA programs are between 30% and 70% of the MPI program.

If we compare the implementation of HTA and MPI with their corresponding serial versions MATLAB$^{\mathrm{TM}}$ and C/Fortran, respectively, the results in Table 1 show that the code sizes of *smv*, *cannon*, *summa* and *lu* increase substantially when going from serial to parallel. The reason is that one-line invocation of the matrix multiply in the serial program is replaced by a whole parallel algorithm. In *jacobi*, *ep*, *mg* and *cg* the same algorithm is implemented in both the serial and parallel version, so the code size doesn't change significantly. Finally, *ft* grows more than other benchmarks when going from serial to parallel, particularly in its MPI version. This increase is because *ft* contains two parallel versions of the same problem: a 1-D version, that partitions along only one dimension of the tridimensional array whose forward and inverse FFT are calculated; and a 2-D version that partitions the array along two dimensions.

It is also worth noting that the time we spent implementing the HTA versions of the programs was much shorter than the time required for implementing the MPI+C/Fortran versions. HTA programs are much easier to write than MPI programs. A functional HTA programs can be obtained trivially by changing array initializations into HTA initializations; performance tuning is mostly accomplished by rearranging indexing in the code in an incremental fashion. Writing an MPI based algorithm requires much more work in planning and thinking, in order to avoid data races, deadlock situations and to get good performance.

6. RELATED WORK

Languages for Parallel Programming have been an object of research for a very long time. Several experimental languages and compilers have been developed so far. Prominent among them is High Performance Fortran [2]. HPF is an extension to FORTRAN that provides new constructs to define the type of data distribution and mapping of data chunks into processors. However, a key drawback in HPF is the inability to operate on a tile (chunk) as a whole. The programmer must explicitly compute the tile sizes and their indexes for each distributed array for each processor. The second drawback is the lack of transparency in communication of data elements across processors. For instance, the main loop in the matrix multiplication program using cannon's algorithm implemented with HPF is shown in Fig. 17. The code in the figure assumes block distribution along both the dimensions of the matrices.

A more closely related work to ours is ZPL [6]. ZPL defines a `region` of a specified shape and size that can be distributed using any specified `distribution` on any specified `grid`. Using the `indexed sequential arrays`, one can build a structure similar to HTA, with operations on tiles as a whole. However, ZPL is still not transparent to the programmer and it is of higher level than HTA. For instance, in ZPL the programmer never knows where and

Table 1: Lines of code in 4 different implementations of each benchmark

Benchmark	*smv*	*cannon*	*jacobi*	*summa*	*lu*	*ep*	*mg*	*ft*	*cg*
MATLAB	6	1	31	1	1	136	1202	114	192
HTA	9	18	41	24	24	168	1500	157	203
C/FORTRAN	100	14	48	14	33	205	1542	261	263
MPI	176	189	364	261	342	242	2443	491	612

```
for i= 1, nprow
  blocksize = n/nprow
  FORALL (j=1:nprow, k=1:npcol)
    j_b = (j-1)*blocksize + 1
    j_e = j_b + blocksize - 1
    k_b = (k-1)*blocksize + 1
    k_e = k_b + blocksize - 1
    c( j_b:j_e, k_b:k_e) = c( j_b:j_e, k_b:k_e) + &
      matmul(a(j_b:j_e, k_b:k_e), b(j_b:j_e, k_b:k_e))
  ENDFORALL
  a = cshift( a, blocksize, 2 )
  b = cshift( b, blocksize, 1 )
enddo
```

Figure 17: HPF - Main loop in Cannon's algorithm.

```
region     R = [1..n, 1..n];
direction  north = [-1, 0]; south = [ 1, 0];
           east  = [ 0, 1]; west  = [ 0,-1];
[R]  repeat
     Temp := (A@north+A@east+A@west+A@south) / 4.0;
     err  := max<< abs(A-Temp);
     A    := Temp;
until err < tolerance;
end;
```

Figure 18: ZPL - Main loop in Jacobi.

how the exchange of data occurs, in a case like *jacobi*. Lack of such a transparency might lead to programs that are difficult to debug. Fig. 18 shows the main part of *jacobi* implementation in ZPL.

7. CONCLUSIONS

In this paper we have presented a novel approach to write parallel programs in object-oriented languages using a class called Hierarchically Tiled Arrays (HTAs). The objects of this class are arrays divided into tiles which may be distributed on a mesh of processors. HTAs allow the expression of parallel computation and data movement by means of indexed assignment and computation operators that overload those of the host language.

We have implemented our new data type as a MATLAB^{TM} toolbox and we have written a number of benchmarks using it. As expected, the benchmarks are easy to read, understand and maintain, particularly when compared to code written using the SPMD programming model. This way, we consider the HTA toolbox to be a powerful tool for the prototyping and design of parallel algorithms that we plan to make publicly available soon.

For many of our benchmarks the performance of HTA code is competitive with that of traditional SPMD codes using MPI; for other benchmarks the HTA code suffers from overhead problems and falls behind in performance. Still, such overheads are related to details of our current implementation and they are not inherent to the HTA approach.

In the current implementation, a client executes the main thread of the program and broadcasts the HTA commands to the servers where the tile resides. Our future work includes improving the scalability of the implementation, possibly using a compiler, and making experiments with more processors. We also want to provide more flexible mapping policies, including the possibility of describing machines with a hierarchical organization of processors onto which to map the tiles. Finally, we plan to replace MATLAB^{TM} in the servers by some freely available open-source software in order to reduce implementation cost.

8. 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. Almasi, L. D. Rose, B. Fraguela, J. Moreira, and D. Padua. Programming for Locality and Parallelism with Hierarchically Tiled Arrays. In *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2003*, volume 2958 of *Lecture Notes in Computer Science*, pages 162–176, College Station, Texas, Oct 2003. Springer-Verlag.

[4] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.

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

[6] B. Chamberlain, S.Choi, E. Lewis, C. Lin, L. Synder, and W. Weathersby. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.

[7] R. A. V. D. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. 9(4):255–274, Apr. 1997.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, USA, 1994.

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

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

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

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

[13] B. Wager. Hyperquicksort: A Fast Algorithm for Hypercubes. In *Hypercube Multiporcessors*, pages 292–299, Philadelphia, PA, 1987. SIAM.