Implementation of Parallel Numerical Algorithms Using Hierarchically Tiled Arrays*

Ganesh Bikshandi[1] Basilio B. Fraguela[2] Jia Guo[1]

María J. Garzarán[1] Gheorghe Almási[3] José Moreira[3] David Padua[1]

Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
{bikshand,jiaguo,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 describe our experience in writing parallel numerical algorithms using Hierarchically Tiled Arrays (HTAs). HTAs are classes of objects 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 development of algorithms with a high degree of parallelism as well as locality. 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}. We have successfully used it to write some widely used parallel numerical programs. The resulting programs are easier to understand and maintain than their MPI counterparts.

1 Introduction

Parallel programs are difficult to develop and maintain. This is particularly true in the case of distributed memory machines, where every piece of data that needs to be accessed by two or more processors must be communicated by means of messages in the program, and where the user must make sure that every machine is working with the latest version of the data. Parallel execution also makes debugging a difficult task. Moreover, more factors influence the performance of parallel programs than that of sequential programs. For example, systems may be

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

heterogeneous; the architecture to consider involves a communication network, and different operating system layers and user libraries may be involved in the passing of a message. As a result, performance tuning is also much harder. The language and compiler community has come up with several approaches to help programmers deal with these issues.

The first approach to ease the burden on the programmer when developing distributed memory programs was based on standard message passing libraries like MPI [8] or PVM [7] which improve the portability of the parallel applications. Data distribution and synchronization must be completely managed by the programmer. The SPMD programming model typically used in conjunction with these libraries leads to unstructured codes in which 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 [11] and UPC [5] improve the readability of the programs by replacing library calls with array assignments, but they still have all the drawbacks of the SPMD approach.

Another strategy to facilitate the programming of distributed memory systems consists of writing the programs as a single thread and letting the compiler take care of the problems of distributing the data and assigning the parallel tasks. This is for example the approach of the High Performance Fortran[9, 10]. Unfortunately, compiler technology is not always capable of generating code that matches the performance of hand-written 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*. HTA operators overload the standard operators of the language. HTA data can be distributed across a collection of servers, and its operations can be scheduled on the appropiate processors. The HTA class provides a flexible indexing scheme for its tiles that allows communication between servers to be expressed by means of simple array assignments. As a result, HTA based programs are single-threaded, which improves readability and ease development and maintenance. Furthermore, the compiler support required by our parallel programming approach is minimal since the implementation of the class itself takes care of the parallelization. In MATLAB^{TM} we have found an ideal platform for the implementation and testing of HTAs. MATLAB^{TM} provides a high-level programming language with object-oriented features that is easy to extend using the *toolbox approach*.

The rest of this paper is structured as follows. HTA syntax and semantics are described in the next Section. Section 3 provides several code examples that use HTA. Section 4 describes the implementation of the HTA language extension as a MATLAB^{TM} toolbox. Section 5 presents our implementation of NAS benchmark kernels using HTA. In Section 6 we analyze two aspects of our approach - performance and programmability. In Section 7 an analytical comparison of HTA with other related languages is given. Finally we conclude in Section 8 with several future directions.

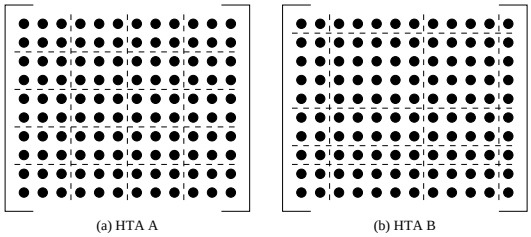

Fig. 1. Two tiled arrays A (a) and B (b).

2 Hierarchically Tiled Arrays: 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. Note that our definition does not require all tiles to have the same size: both HTAs in Fig. 1 are legal.

2.1 Dereferencing the contents of an HTA

Given an arbitrary HTA, there are two basic ways to address its contents: *hierarchical* or *flat*. Hierarchical addressing identifies a subset of the HTA at the top level of the hierarchy, then identifies subsets of those elements at lower levels of the hierarchy, and so on down to possibly the lowest level.

Flat addressing disregards the hierarchical nature of the HTA and addresses elements by their absolute indices, as in a normal array. Flattening can be applied to any level of the hierarchy. Thus, a series of intermediate possibilities lie between hierarchical and flat addressing.

As an example, consider HTA `A` shown in Fig. 1(a). The element in its fifth row and sixth column can be referenced using flat addressing as `A(5,6)`. The same element can be referenced using hierarchical addressing as `A{3,2}(1,3)`, where curly brackets are used to index tiles and parentheses are used to index the elements in the bottommost tile. Indexing using triple notation `init:step:limit` is also provided for both hierarchical and flattened addressing.

2.2 HTA arithmetic operations

The semantics of HTA arithmetic operations depend on the operands. The following are the types of operations and the resulting semantics.

- HTA $\oplus$ Scalar: In this case, the scalar is promoted to an HTA with a single tile whose value is just a scalar and operated with each tile of the HTA.
- HTA $\oplus$ Matrix: In this case also, the matrix is promoted to an HTA with a single tile and operated with the HTA on the left. However, unlike the scalar promotion, matrix promotion requires a legality check. The condition

for legality of operations depend on the operator - an addition or a subtraction requires the matrix to match the shape of each tile of the HTA, while multiplication requires the number of columns of the matrix to be same as that of rows of each tile of the HTA.

- HTA $\oplus$ HTA: Two cases are distinguished here. The first one involves two HTAs with the *same topology* - with the same number of tiles in every dimension at the top level and the corresponding tiles being legally operable. The resulting HTA has the same topology as the input ones, with each of its tiles associated to the computation of the corresponding input tiles. The other case involves HTAs in which one of them as a whole has the same topology as each one of the tiles of the other one at a given level of subdivision. In this situation, the semantics is similar to that of HTA $\oplus$ Matrix.

2.3 HTA Assignments

The semantics for assignments to HTAs are similar to 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 if the HTA resulting from the 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) one is legal. When the right HTA has a single tile, it is replicated in each one of the tiles selected in the left HTA.

2.4 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, the HTAs `A` and `B` from Fig. 1 could be created as follows, using a source matrix `MX`:

```
A = hta(MX, {1:2:10,1:3:12});
B = hta(MX, {[1,2,6,8,9], [1,3,8,12]});
```

The HTAs built above are local. In order to request the distribution of the top-level tiles of the HTA on a mesh of processors, the last argument of the constructor must be a vector specifying the shape of the mesh. The distribution is currently fixed to be block-cyclic.

HTAs can also be built as structures whose tiles are empty. In this case the constructor is called just with the number of tiles desired in each dimension. The empty tiles can be filled in later by means of assignments. As an, example, the following statement generates an empty 4×4 HTA whose tiles are distributed on a 2×2 processor mesh:

```
A = hta(4, 4, [2, 2]);
```

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

Fig. 2. Main loop in Cannon's algorithm.

3 Parallel Programming using HTAs

Codes using HTAs have a single thread of execution that runs in a client (master). The client is connected to a distributed memory machine with an array of processors, called servers. HTAs can be created on the client side and their tiles can be distributed to the severs. An operator involving HTA requires the client to broadcast the operation to the servers where they are executed in parallel. Some operators, once initiated by the client, involve communication among severs without the further involvement of the client. Some other operators involve accessing arbitrary tiles across different severs, mediated by the client. We will see some of these common examples in this section.

The first example we consider is the Cannon's matrix multiplication algorithm shown in Fig. 2. Cannon's matrix multiplication algorithm [4] is an example of code that requires communication between the servers. In our implementation of the algorithm, the operands, denoted `a` and `b` respectively, are HTAs tiled in two dimensions. The HTAs 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 tiles 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 local matrix multiplication, the tiles of `b` and `a` are circularly shifted in the first and second dimensions respectively. 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 lower 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. The statement $c{=}a{*}b$, in Fig. 2, initiates the above mentioned parallel computation with circular shift after each iteration. The circular shift is done by `circshift`, an HTA-overloaded version of the native `circshift` in MATLAB^{TM} . In this code the function involves communication because the HTAs are distributed.

The next example shows how to reference arbitrary elements of HTAs. The blocked Jacobi relaxation code in Fig. 3 requires a given element to compute its new value as a function of the values of its four neighbors. Each block is represented by a tile of the HTA `v`. In addition the tiles also contain extra rows and columns for use as border regions exchanging information with the neighbors. Border exchange is executed in the first four statements of the main loop. The actual computation step (last statement in the loop) uses only local data.

```
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
```

Fig. 3. Parallel Jacobi relaxation

```
input = hta(px, py, [px py]);
input{:, :} = zeros(px, py);
output = parHTAFunc(@randx, input);
```

Fig. 4. Filling an HTA with uniform random numbers

Easy programming of embarrassingly parallel and MIMD style codes is also possible using HTAs thanks to the `parHTAFunc` 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 its arguments. At least one of these arguments must be a distributed HTA. The function `func` will be executed over each of the tiles, on their owning servers. 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.

Fig. 4 illustrates the usage of `parHTAFunc` to fill an array represented as HTA with uniform random numbers in the range [0,1]. A distributed HTA `input` with one tile per processor is built and is sent as the argument to `randx` via the `parHTAFunc`. The function `randx` is similar to `rand` in MATLAB^{TM} but generates a different sequence on each processor. Specific details of this function is ignored for brevity. 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 random numbers filled in.

4 Execution Model and 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 3 main reasons: wide user base, support of polymorphism and extensibility as a toolbox. HTAs are implemented as a new type of object, accessible through the constructors described in Section 2.

The HTA class methods have been written either in MATLAB^{TM} or in C that interfaces with MATLAB^{TM} (version R13 or above) through MEX. In general,

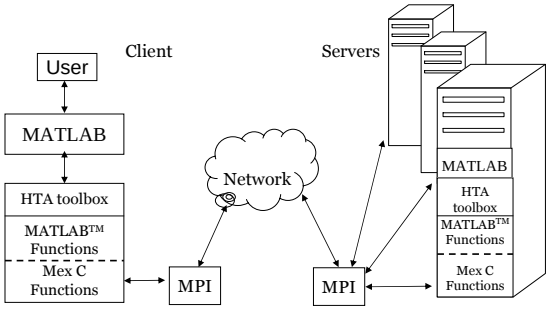

Fig. 5. HTA implementation in MATLAB^{TM}

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. Communications between the client and the servers are implemented using the MPI [8] library, thus methods that involve communication were written in C in order to use the message-passing library.

The architecture of our MATLAB^{TM} implementation is shown in Fig. 5. 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. The lower layers of the HTA toolbox deal with communication, while the higher layers implement the syntax.

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.

When an HTA is distributed, the client keeps an image of its complete structure, so that legality checks can always be made locally. As the number of servers grows, the centralized legality check and the required broadcast of commands from the client will become a bottleneck. Thus HTA implementations for highly-parallel systems should be based on compilers in order to scale as required.

5 NAS Benchmark Suite

In this section we present our experience with writing highly parallel numerical programs using HTA. Specifically, we deal with the kernels of the NAS benchmarks [1]. We have successfully implemented four kernels so far - *cg*, *ep*, *mg* and *ft*. We are also in the process of implementing the other kernels. We first created serial versions of the programs in MATLAB^{TM} and incrementally parallelized them with minimal efforts using the HTA representation. In the following subsections we present a detailed description of *ep*, *mg* and *ft*.

5.1 NAS *Kernel ep*

The NAS kernel *ep* is an embarrassingly parallel application that generates pairs of uniformly distributed pseudo-random numbers and tabulates those pairs that lie in successive square annuli. In the parallel implementation of the algorithm, each processor generates its section of random numbers and does the above mentioned computation locally and exchange the results through a reduction at the end. This is implemented using the `parHTAFunc` described in Section 3. The resulting code looks similar to that shown in Fig. 4, with additional computations that involves finding the number of pairs that lie in the successive square annuli. All these steps are encoded inside the `randx` function of the Fig. 4 that is called via the `parHTAFunc`. Function `myHTARank` is used to identify the rank of the servers in execution, inside `randx`.

5.2 NAS *Kernel mg*

The NAS kernel *mg* solves the Poisson's equation, $\nabla^2 u = v$, in 3D using a multigrid V-cycle. An outline of the algorithm, as presented in [1] is shown in Fig. 6. P, Q, A and S are arrays of coefficients. Each iteration of the algorithm consists of two steps, 1 and 2, listed in Fig. 6. The step 2 is recursive and is also explained in the same figure. The step 1 consists of finding the residual $r = v - Au$. The step 2 consists of solving the residual equation $Ae = r$ and updating the current solution u, $u = u + e$. Solving the residual equation is done using the V-cycle approach, described as M^k, in the figure. The first step of the V-cycle is restricting (`restrict`) larger grids to smaller ones (step 3) successively (step 4), until we reach the smallest grid that could be solved trivially (step 8). Then the reverse process of interpolation (`interpolate`) takes place where a smaller grid is expanded to a larger grid (step 5). The above sequence is repeated for several iterations until the solution converges.

The grids `v` and `u` are 3-dimensional and are represented as 3-dimensional arrays. $r_i, 0 < i < k$, is an array of grids with the k^{th} grid being the largest. Each grid is 3-dimensional and is represented in the same way as u. P, Q, A and S are banded diagonal matrices with the same 4 values in each row. Hence they are represented using a vector with 4 values. Given the above representations, it is straight forward to implement the above algorithm in MATLAB^{TM} . In order to parallelize this algorithm, the grids must be divided into several chunks along each dimension and each such chunk must be allocated to a processor, using a processor map. After each operation, communication of the boundary values in each dimension should be performed. Shadow regions are allocated in each chunk to hold the boundary values.

In a typical MPI program, the programmer is responsible for doing all the above mentioned steps. The algorithm is implemented in parallel using the HTA representation in an easy manner as follows. The grids `v` and `u` are represented as a 3-D HTAs of size $dx \times dy \times dz$. r_i is represented as an array of HTAs, with each HTA representing a grid of size i. Fig. 7 presents an outline of this step. Also shown in the figure is the allocation of extra space for shadow regions in each

(1) $r = v - A\,u$
(2) $u = u + M^k\,r$

where M^k is defined as follows:
$z_k = M^k\,r_k$:
if $k > 1$
(3) [`restrict`] $r_{k-1} = Pr_k$
(4) [`recursive solve`] $z_{k-1} = M^{k-1}\,r_{k-1}$
(5) [`interpolate`] $z_k = Qz_{k-1}$
(6) [`evaluate residue`] $r_k = r_k - A\,z_k$
(7) [`smoothen`] $z_k = z_k + S\,r_k$
else
(8) $z_1 = S\,r_1$
end

Fig. 6. *mg* - Outline of the algorithm

```
%px, py, pz are processors along X, Y and Z axes
for i = 1 : k
%add shadow regions for the boundaries
    sx = nx(i) + 2 * px;
    sy = ny(i) + 2 * py;
    sz = nz(i) + 2 * pz;
    r{i} = hta(zeros(sx, sy, sz), {1 : sx/px : sx, 1 : sy/py : sy, 1 : sz/pz : sz}, [px py pz]);
end
u = hta(r{k});
v = hta(r{k});
```

Fig. 7. *mg* - Creation of HTAs

tile. It should be noted that the HTA constructor automatically maps the tiles to the processors according to the specified topology as explained in Section 2.4. The next important step is the communication of the boundary values, shown in Fig. 8. The figure shows the communication along X dimension; the communications along the other dimensions look very similar. The grid operations are implemented as a subroutine with the HTAs as parameters. One such operation, `restrict`, is shown in the Fig. 9. The above subroutine implemented as such is inefficient, as each addition operation requires the client to send a message to the servers along with the two operands. This is mitigated by the use of `parHTAFunc`, explained in section 3, as follows - $s = parHTAFunc(@restrict, r, s)$. This requires only one message to be sent to the servers. Thus `parHTAFunc`, apart from facilitating MIMD-style of programming, also enables optimization. Finally, the program involves computing the sum $\sum {r_k}^2$, which is done using the HTA-overloaded parallel `sum` and `power` operators.

%dx is the size of the HTA along X dimension

$u\{1:dx-1,:,:\}(n1,2:n2-1,2:n3-1) = u\{2:dx,:,:\}(2,2:n2-1,2:n3-1);$

$u\{dx,:,:\}(n1,2:n2-1,2:n3-1) = u\{1,:,:\}(2,2:n2-1,2:n3-1);$

$u\{2:dx,:,:\}(1,2:n2-1,2:n3-1) = u\{1:dx-1,:,:\}(n1-1,2:n2-1,2:n3-1);$

$u\{1,:,:\}(1,2:n2-1,2:n3-1) = u\{dx,:,:\}(n1-1,2:n2-1,2:n3-1);$

Fig. 8. *mg* - Communication of Boundary Regions

%i1b,i2b,i2e - begining position in tiles of r along X,Y,Z respectively;
%i1e,i2e,i3e - ending position in tiles of r along X,Y,Z respectively;
%m1j,m2j,m3j - ending position in tiles of s along X,Y,Z respectively;

$s\{:,:,:\}(2:m1j-1,2:m2j-1,2:m3j-1) = 0.5D0 * r\{:,:,:\}(i1b:2:i1e,i2b:2:i2e,i3b:2:i3e)...$

$+0.25D0 * (r\{:,:,:\}(i1b-1:2:i1e-1,i2b:2:i2e,i3b:2:i3e)...$

$+r\{:,:,:\}(i1b+1:2:i1e+1,i2b:2:i2e,i3b:2:i3e)...$

$+r\{:,:,:\}(i1b:2:i1e,i2b-1:2:i2e-1,i3b:2:i3e)...$

$+r\{:,:,:\}(i1b:2:i1e,i2b+1:2:i2e+1,i3b:2:i3e)...$

$+r\{:,:,:\}(i1b:2:i1e,i2b:2:i2e,i3b-1:2:i3e-1)...$

$+r\{:,:,:\}(i1b:2:i1e,i2b:2:i2e,i3b+1:2:i3e+1));$

+ several other similar operations, ignored here for brevity

Fig. 9. *mg* - Operation `restrict` (Line 3 of Fig. 6)

5.3 NAS Kernel *ft*

NAS kernel *ft* numerically solves certain partial differential equations (PDE) using forward and inverse Fast Fourier Transform. Consider the PDE, $\frac{\partial u(x,t)}{\partial t} = \alpha\nabla^2 u(x,t)$ where x is a position in 3-dimensional space. Now applying FFT on both side we get, $\frac{\partial v(z,t)}{\partial t} = -4\alpha\pi^2|Z|^2 v(z,t)$, where $v(z,t)$ is the Fast Fourier transform of $u(x,t)$. This has the solution $v(z,t) = e^{-4\alpha\pi^2|z|^2 t}v(z,0)$. Thus, the original equation can be solved by applying the FFT to u, then multiplying the result by a certain exponential, and finding the inverse FFT of the result.To implement this algorithm we essentially need to have a forward and inverse FFT operator. Fortunately, MATLAB^{TM} already has this operator.

In the parallel version of FFT for N-Dimensional arrays, one of the dimensions is not distributed. 1D-FFTs are calculated along each non-distributed dimension, one by one. If a dimension is distributed, it is transposed with a non-distributed dimension and the 1-D FFT is applied along that dimension. In the MPI version the programmer has to implement the transpose using `alltoall` communication and cumbersome processor mapping data structures. An implementation of this algorithm for calculating the forward FFT of a 3-D array, 2-D decomposed along Y and Z axes, using HTAs is shown in Fig. 10. The function `fft` is the native MATLAB^{TM} function to calculate the FFT of an array along a specified dimension. The operator `dpermute` permutes the data of the `fft` array without changing its underlying structure. In this kernel, the HTAs are of complex data type.

$x = hta(complex(nx, ny, nz), \{1, 1 : ny/py, 1 : nz/pz\}, [1\ py\ pz]);$
$x = parHTAFunc(@compute\_initial\_conditions, x);$
$x = parHTAFunc(@fft, x, [\,], 1);$
$x = dpermute(x, [2\ 1\ 3]);$
$x = parHTAFunc(@fft, x, [\,], 1);$
$x = dpermute(x, [3\ 2\ 1]);$
$x = parHTAFunc(@fft, x, [\,], 1);$

Fig. 10. *ft* - Calculating Forward FFT using HTA

6 Analysis of the Results

In this subsection we discuss two aspects of our implementation of NAS benchmark kernels using HTA - Performance and Programmability. We conducted our experiment in an IBM SP system consisting of two SMP nodes of 8 Power3 processors running at 375 MHz and sharing 8 GB of memory each. In our experiments we allocated one processor for the client (master) and either other 4 or 8 additional ones for the servers. In both cases, half of the processors were allocated from each one of the two SMP nodes. The next natural step of experimentation involves 16 servers, but that would require 17 processors. The C files of the toolbox were compiled with the VisualAge C xlc compiler for AIX, version 5.0, while the MPI versions of the NAS benchmarks were compiled with the corresponding xlf compiler from IBM. In both cases the `O3` level of optimization was applied. The computational engine and interpreter for the HTA was MATLAB^{TM} R13, while the MPI programs used the native highly optimized IBM ESSL library to perform their computations. The MPI library and environment in both cases was the one provided by the IBM Parallel Environment for AIX Version 3.1.

6.1 Performance

Fig. 11 is the execution time and speedup plot using 4 and 8 servers (processors) for both MPI and HTA programs . *ft_1d* and *ft_2d* correspond to *ft* with 1-D and 2-D decomposition respectively. Also shown in the figure is the speedup for *mg* without the use of `parHTAFunc` (*mg_H*). 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 kernel *ep* 536870912 random numbers are generated. All of them correspond to Class A of the NAS benchmark. The raw execution time is considerably larger for HTA - the HTA system is built over MATLAB^{TM} which is a slow interpreted environment. However, the speedup obtained for each kernel is significant.

Kernel *ep* has a perfect speedup, as it is embarrassingly parallel and there is no communication overhead apart from the initial distribution and final reduction. Kernel *ft* has a super-linear speedup for HTA version. In *ft* the timing also includes the initialization of the complex array with random entries. This takes huge time in a serial MATLAB^{TM} program, which we believe is due to overhead

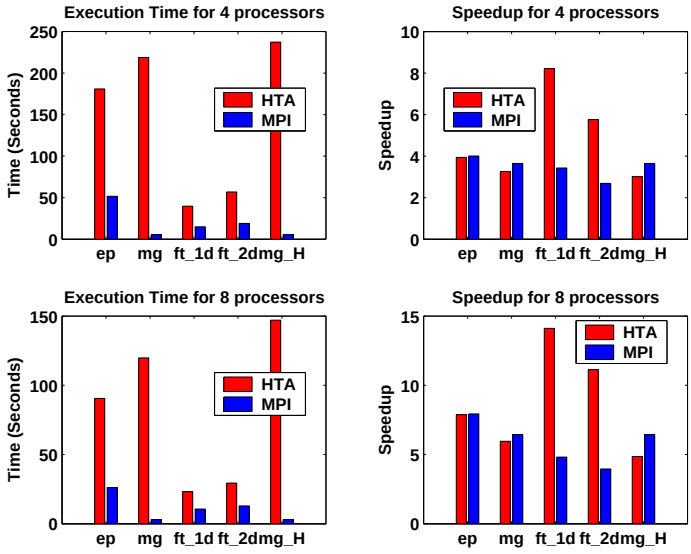

Fig. 11. Execution Time and Speedup

in cache or TLB. For kernel *mg* the speedup of HTA program is slightly lesser than that of MPI program. We discovered two main sources of overhead that are the reasons for this. The first overhead is associated with the interpretation of every command and its broadcast to the servers each time. We plan to have a compiler support to overcome this overhead. The other overhead is caused by the need to make a full copy of the LHS of any indexed HTA assignment operation. This is due to the fact that the (overloaded) indexed assignment operator cannot deal with arguments passed in by reference; it needs arguments to be passed by value. This is a limitation imposed by our need to operate inside the MATLAB^{TM} environment. This overhead is pronounced in *mg*, which has a lot of communication. Readers should note that the NAS benchmark kernels are highly optimized, while our current HTA versions are not that optimized.

6.2 Ease of programming

The primary goal of our research is to ease the burden of parallel programmers. In order to show the effectiveness of HTA for this purpose, we measure the number of lines of key parts of the HTA and MPI programs. Though the number of lines of code is not a standard metric to measure the ease of programming, it gives a rough idea about the complexity of programs.

Fig. 12 shows the number of lines of code for two main parts of *mg* and *ft* programs - data decomposition/mapping and communication. Kernel *ep* look very similar in both MPI and HTA versions, resulting in equal sized codes in both the cases. In *mg*, the domain decomposition/mapping has almost same number of lines because both the HTA and MPI programs do similar steps to find the

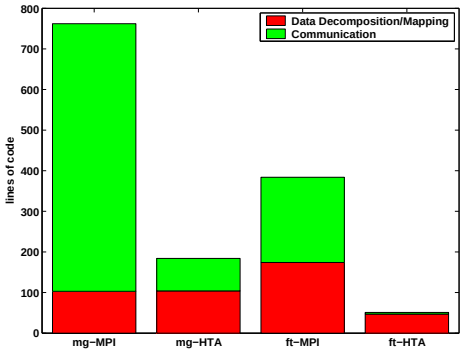

Fig. 12. Lines of code for key sections of *mg* and *ft*.

init and limit of each array chunk. However, the MPI version of *ft* does additional computations for the later transpose operation and hence it is bigger than the HTA version. The number of lines consumed by the communication operations is significantly lesser in the HTA programs for both *mg* and *ft*. In the HTA programs, communication consists of simple assignments for *mg* as explained in Fig. 8 and an overloaded HTA-aware `permute` operation for *ft* as explained in Fig. 10. Another metric worth measuring is the degree of similarity of parallel programs to the serial versions. HTA programs look very similar, in style and size, to the serial programs written in MATLAB^{TM} , while MPI programs look drastically different from their serial versions. Readers should note that NAS benchmarks are very well written and are easily readable; programs written by less experienced MPI programmers are more complex.

7 Related Works

Languages for Parallel Programming have been an object of research for 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 would look like that in Fig. 13. 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

```
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
```

Fig. 13. 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;
```

Fig. 14. ZPL - Main loop in Jacobi.

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 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 14 shows the main part of *jacobi* implementation in ZPL.

8 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). 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. HTAs improve the ability of the programmer to reason about a parallel program, particularly when compared to the code written using SPMD programming model. HTA tiling can also be used to express memory locality in linear algebra routines.

We have implemented our new data type as a MATLAB^{TM} toolbox. We have successfully written three NAS benchmark kernels in the MATLAB^{TM} +

HTA environment. The kernels are easy to read, understand and maintain, as the examples shown through our paper illustrate. We discovered two key overheads suffered by HTA codes, both related to details of our current implementation. First, the current implementation combines the interpreted execution of MATLAB^{TM} with the need to broadcast each command to a remote server. This could be mitigated in the future by more intelligent ahead-of-time broadcasting of commands or by the deployment of a compiler. The second cause of overhead is the need to use intermediate buffering and to replicate pieces of data. This is not because of algorithmic requirements, but due to the need to operate inside the MATLAB^{TM} environment. This effect can be improved by more careful implementation.

In summary, we consider the HTA toolbox to be a powerful tool for the prototyping and design of parallel algorithms and we plan to make it publicly available soon along with the benchmark suite.

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. De Rose, B.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*, to be published in Lecture Notes in Computer Science, vol. 2958, College Station, Texas, Oct 2003. Springer-Verlag.
4. L.E. 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.L. Chamberlin, S.Choi, E.C. Lewis, C. Lin, L. Synder, and W.D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.
7. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.
8. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.
9. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling fortran d for mimd distributed-memory machines. *Commun. ACM*, 35(8):66–80, 1992.
10. C. Koelbel and P. Mehrotra. An overview of high performance fortran. *SIGPLAN Fortran Forum*, 11(4):9–16, 1992.
11. R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.