Programming for Parallelism and Locality with Hierarchically Tiled Arrays

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

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

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

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

Abstract

Tiling has proven to be an effective mechanism to develop high performance implementations of algorithms. Tiling can be used to organize computations so that communication costs in parallel programs are reduced and locality in sequential codes or sequential components of parallel programs is enhanced.

In this paper, a data type - Hierarchically Tiled Arrays or HTAs - that facilitates the direct manipulation of tiles is introduced. HTA operations are overloaded array operations. We argue that the implementation of HTAs in sequential OO languages transforms these languages into powerful tools for the development of high-performance parallel codes and codes with high degree of locality. To support this claim, we discuss our experiences with the implementation of HTAs for MATLAB and C++ and the rewriting of the NAS benchmarks and a few other programs into HTA-based parallel form.

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

General Terms Language

Keywords Parallel programming, data-parallel, locality enhancement, tiling

1. Introduction

This paper introduces a type of object that facilitates the provision of locality and parallelism. These objects are tiled arrays whose tiles could be tiled and whose components, tiles or the underlying scalars, can be accessed using an intuitive notation. In other words, a hierarchically tiled array can be accessed as a flat n-dimensional array where the scalar elements are addressed using the conventional subscript notation or as a hierarchy of tiles whose components are addressed by a sequence of subscript tuples, one for each level of tiling.

The main motivation behind the design of these *Hierarchically Tiled Arrays* or *HTA*s is that, for a wide range of problems, data tiling has proven to be an effective mechanism for improving performance by enhancing locality [26] and parallelism [3, 9, 8, 21]. Our objective in this paper is to make a first attempt in the identification of the set of operations on tiles needed to develop programs that are at the same time readable and efficient and to show that such operations facilitate programming of high performance computations.

To implement parallel computations for distributed memory machines, HTA tiles are distributed. Parallel computations and communications are represented by overloaded array operations. Thus, interprocessor communication operations are represented by assignments between HTAs with different distributions. Parallel computations take the form of array (or data parallel) operations on the distributed tiles. HTAs are designed for use by single threaded programs where parallel computations are represented as array operations. As a result, parallelism is highly structured, which improves readability over the SPMD paradigm. The use of array operations to represent parallel computations on a distributed memory system is, of course, not new. It was the only mechanism to express parallelism in Illiac IV [5] and other SIMD systems and it has been used in a variety of languages including High Performance Fortran [3] and its variants [10] and ZPL [9], X10 [11], and others. The main contribution of HTA is the use of array operations to manipulate tiles directly. HTAs are also useful for the development of efficient programs for shared-memory and uniprocessor computations because data tiling tends to reduce data traffic in SMP computations and increase locality in sequential computations. More specifically, HTAs are a convenient notation to program multicore processors and hybrid parallel machines such as the CELL Broadband Engine [22]. Furthermore, because of their hierarchical nature, HTAs can be used to take advantage of the multiple levels of parallelism that can be exploited by systems such as clusters of multicore processors as well as multiple levels of memory hierarchy.

The rest of the paper is organized as follows. In Section 2, we describe HTAs and the operations on them that we have found most useful in our studies. In Section 3, we present some code examples to illustrate how a few important parallel kernels can be implemented with HTAs. Our objective in this Section is to demonstrate that HTAs can be used to produce elegant and intuitive formulations of these kernels. In Section 4 we discuss our efforts to incorporate HTAs in two sequential languages: MATLAB and C++. Finally, in Section 5 we compare the HTA approach with other parallel programming approaches. Finally, Section 6 concludes.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

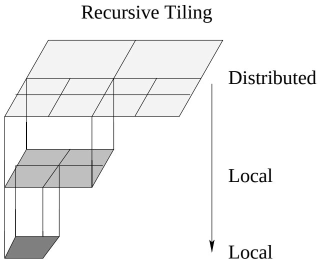


Figure 1. Pictorial view of a Hierarchically Tiled Array.

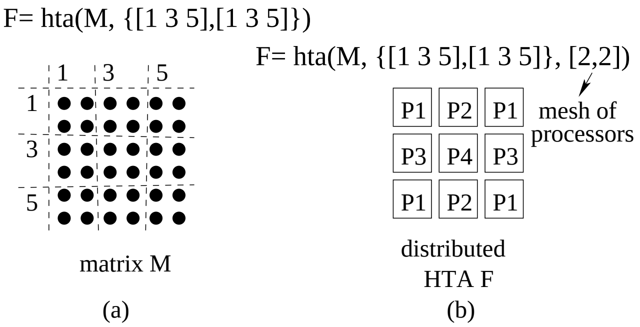


Figure 2. Construction of an HTA by partitioning an array-(a). Mapping of tiles to processors-(b).

2. A New Programming Paradigm

In this Section we describe the semantics of the Hierarchically Tiled Arrays (HTA) (Section 2.1), their construction and distribution (Section 2.2) mechanisms to access the HTA components (Section 2.3), assignment statements and binary operations (Section 2.4) and other HTA methods (Section 2.5). For simplicity, in all the examples in this Section we use the syntax of our MATLAB extension.

2.1 Semantics

Hierarchically Tiled Arrays (HTAs) are arrays partitioned into tiles. These tiles can be either conventional arrays or lower level Hierarchically Tiled Arrays. Tiles can be distributed across processors in a distributed-memory machine or be stored in a single machine according to a user specified layout. We distribute the outermost tiles across processors for parallelism and utilize the inner tiles for locality. Figure 1 shows an example of an HTA with two levels of tiling.

2.2 Construction of HTAs

Two different approaches can be used to create an HTA. The first is to tile an existing array using delimiters for 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, as show in Figure 2-(a). The second parameter of the HTA constructor 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.

Alternatively, an HTA can be built as an empty set of tiles. In order to create an HTA of this form, the HTA constructor is invoked with the number of desired tiles per dimension. For example, `hta(3,3)` creates an HTA with 3×3 empty tiles. To complete the HTA, each tile must be assigned a content after the empty shell is created.

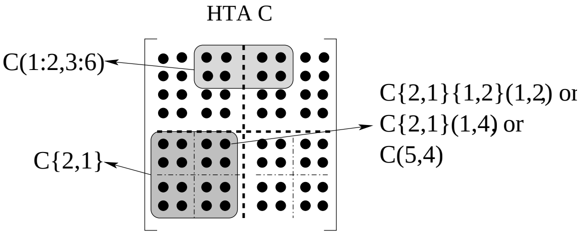


Figure 3. Accessing the contents of HTAs.

The tiles of an HTA can be local or distributed across processors. To map tiles to processors, the topology of the mesh of processors and the type of distribution (block, cyclic, block cyclic, or a user-defined distribution) must be provided. Figure 2-(b) shows an example where a 6×6 matrix is distributed on a 2×2 mesh of processors. The last parameter of the HTA constructor specifies the processor topology. In our current implementation, the default distribution is a cyclic distribution of the tiles on the mesh, which corresponds to a block cyclic [3] distribution of the matrix contained in the HTA, with the blocks defined by the topmost level of tiling.

Notice that, although not illustrated here due to space limitations, HTAs can be built with several levels of tiling, like those shown in Figure 1.

2.3 Accessing the Components of an HTA

Figure 3 shows examples of how to access HTA components. The expression `C{2,1}` refers to the lower left tile. The scalar element in the fifth row and fourth column can be referenced as `C(5,4)` just as if `C` were an unpartitioned array. This element can also be accessed by selecting the leaf 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)`. Regions such as `C(1:2,3:6)` can also be accessed using parenthesis to disregard the tiling of the HTA. The result of such expressions do not keep the tiled structure of the HTA, that is, `C(1:2,3:6)` will simply return a plain standard 2×4 matrix. If the HTA `C` is distributed, this output matrix is a replicated local object that appears in all the processors. Flattening is particularly useful when transforming a conventional program onto a tiled form for locality/parallelism or both. During the intermediate steps of the transformation, regions of the program can remain unmodified and arrays accessed as if they were not partitioned while in other regions, the arrays are manipulated by tiles.

Sets of components may be chosen at any level and along each dimension using triplets of the form *begin*:*step*:*end*. The : notation can be used in any index to refer to the whole range of possible values for that index. For example, `C{2,:}(1:2:4,:)` refers to the odd rows of the two lower outer-level tiles of `C`.

We can also use boolean arrays as HTA subscripts When this logical indexing is applied every true element in the boolean array used as a subscript designates a tile of the HTA based on the position of the true elements. As illustrated in Figure 4 logical indexing allows the selection of arbitrary, banded diagonal or upper triangular tiles of an HTA.

2.4 Assignments and Binary Operations

We generalize the notion of conformability of Fortran 90. When two HTAs are used in an expression, they must be conformable.

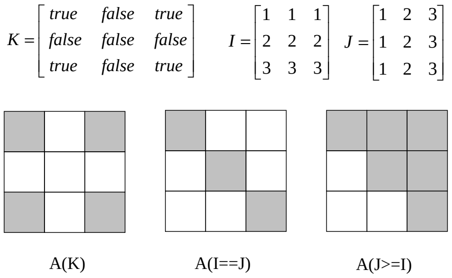

$K = \begin{bmatrix} true & false & true \\ false & false & false \\ true & false & true \end{bmatrix}$
$I = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$
$J = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$
A(K)
A(I==J)
A(J>=I)

Figure 4. Logical indexing in HTA.

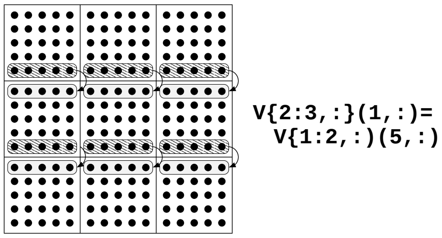


Figure 5. Assignment of all the elements in the last row of each one of the tiles located in the rows of tiles 1 and 2 to the first row of the corresponding tiles in the rows of tiles 2 and 3.

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 the operation to act on them. The operation is executed tile by tile, and the output HTA has the same topology as the operands.

Also, an HTA and an untiled array are conformable when the array is conformable with each of the leaf tiles of the HTA. An HTA and a scalar are always conformable. When an untiled array is operated with an HTA, each leaf tile of the HTA is operated with the array. Also, when one of the operands 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.

Assignments to HTAs must follow similar rules to those of binary operators. 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).

References to local HTAs do not involve communication. However, in distributed HTAs assignments between tiles which are in different processors involve communication. Consider a 3×3 distributed HTA, `V`. The assignment `V{2:3,:}(1,:) = V{1:2,:}(5,:)` copies all the elements in the fifth row in the two first rows of tiles to the first row in the tiles in the two last rows of tiles as shown in Figure 5. When the tiles of `V` are distributed across processors, this assignment involves communication.

2.5 Other HTA Methods

We have overloaded frequently-used functions on arrays such as `circshift`, `transpose`, `permute`, or `repmat`, as well as the standard arithmetic operators so that when applied to HTAs they operate at the tile level. For example, the MATLAB function `circshift` implements circular shifts for arrays. The overloaded HTA version shifts instead whole tiles of HTAs, which involves interprocessor communication when the HTAs are distributed.

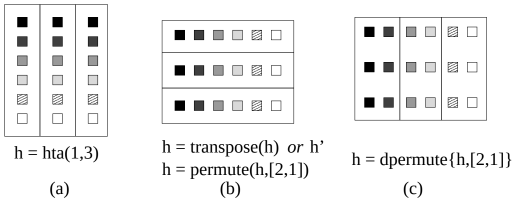
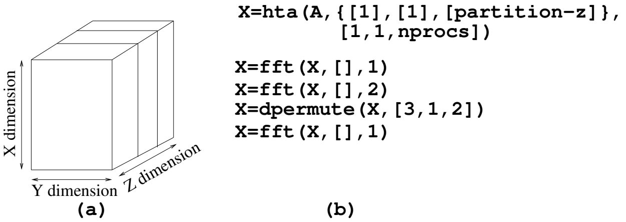


Figure 6. Permute and dpermute.



```
X=hta(A,{[1],[1],[partition-z]},
        [1,1,nprocs])

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

Figure 7. Data Permutation in FFT.(a)-Pictorial view.(b)-code

We have also implemented methods, only applicable to the HTA class, that we have found to be useful in many parallel programs. These are methods like `reduceHTA` which is a generalized reduction method that operates on HTA tiles, and `parHTA` which applies in parallel the same function to each tile of an HTA.

3. HTA operations

In this Section, we use code examples to illustrate how the HTA assignments and methods mentioned in the previous section can be used to write parallel programs. The examples are simple kernels and the NAS parallel benchmarks [2], which we implemented using MATLAB extensions. We classify the methods as either communication operations or global computations.

3.1 Communication Operations

In HTA programs, communication is represented as assignments on distributed HTAs as shown in the example in Figure 5. However, in HTA programs, communications can also be expressed using methods such as `permute`, `circshift` and `repmat`. We discuss them in the next Sections.

3.1.1 Permute Operations

Figure 6 shows two examples of `permute` operations. Figure 6-(b) shows the HTA that results after applying the overloaded MATLAB `transpose` or `permute` operator to the HTA in Figure 6-(a). Figure 6-(c) shows the HTA that results after applying a new method called `dpermute`, which is a special case of a permutation operation where only the data are transposed, but the shape of the containing HTA remains the same. That is, the number of tiles in each dimension remains constant. Thus, as shown in Figure 6-(c), after the `dpermute` operator the data have been transposed, but the resulting HTA contains 1×3 tiles, as in Figure 6-(a).

The `dpermute` operator is applied in the NAS FT, and operates on a 3-D array (`A`) which is partitioned into tiles which are distributed along the `Z` dimension, as shown in Figure 7-(a). To compute the Fourier Transform (FT) along the `Z` dimension we need to bring the blocks from the distributed dimension to the undistributed ones so that the FT can be locally applied. Figure 7-(b) shows an outline of the NAS FT. The FT along the first and second dimension of an HTA is computed using the overloaded version of the standard MATLAB `fft` operator which applies the standard MATLAB `fft`

```
function C = cannon(A,B,C)
  for i=2:m
    A{i,:} = circshift(A{i,:}, [0, -(i-1)]);
    B{:,i} = circshift(B{:,i}, [-(i-1), 0]);
  end
  for k=1:m-1
    C = C + A * B;
    A = circshift(A, [0, -1]);
    B = circshift(B, [-1, 0]);
  end
```

Figure 8. Cannon's algorithm using HTAs.

```
function C = matmul (A, B, C)
  if (level(A) == 0)
      C = C + A * B;
  else
    for i=1:size(A,1)
      for k=1:size(A,2)
        for j=1:size(B,2)
          C{i, j} = matmul(A{i,k}, B{k,j}, C{i,j});
        end
      end
    end
  end
```

Figure 9. Recursive matrix-matrix multiplication that exploits cache locality.

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 explained above. Notice that when programming with HTAs it is possible to determine where communication occurs if the distribution of tiles across the processors is known.

3.1.2 Circular Shift

The communication pattern of the circular shift (circshift) operator appears in Cannon's algorithm [7], which is shown in Figure 8. Here A and B are $m \times m$ HTAs tiled along both dimensions and mapped onto a mesh of $n \times n$ processors. To implement Cannon's algorithm, the first loop shifts circularly the tiles in A and B to place them in the appropriate position. To this end, tiles in row i of A are circularly shifted *i-1* times to the left. Similarly, tiles in column i of B are circularly shifted up *i-1* times. In each iteration of the main loop, each server executes a local matrix-matrix multiplication of the tiles of A and B which each processor owns. The result is accumulated into a local tile of the resulting HTA, C. Then tiles of A and B are circularly shifted once. Tiles of A are circularly shifted to the left, and tiles of B are circularly shifted up. At the end of the main loop, the result of the matrix multiplication is the HTA C which is distributed across processors with the same mapping of A and B. A more conventional implementation of Cannon's algorithm will shift rows of matrix A and columns of the matrices B instead of shifting tiles and the multiplication will be element by element, not a matrix-matrix multiplication of tiles as in our HTA implementation. (The MATLAB * operator has been overloaded such that it performs a tile-by-tile matrix multiplication). The main advantages of the tiled approach is aggregation of data into a tile for communication and the increased locality resulting from a single matrix-matrix multiplication over the element by element multiplication. We can further increase cache locality by using HTAs with two or more levels of tiling. Thus, the tiled matrix-matrix multiplication matmul of Figure 9 can be applied to each pair of corresponding tiles by writing C = parHTA (@matmul, A, B, C) in the Cannon's algorithm of Figure 8. By using parHTA (discussed in Section 3.2), the matmul function is applied in parallel to all the tiles of HTAs A, B and C. In Figure 9 the level(A) function will return 0 when A is either a scalar or a matrix. If A is not a scalar or a matrix we have to recursively proceed down into the HTA hierarchy until we reach the leaf tile of the HTA. The function size(H,i) returns the number of tiles of an HTA H along the i-th dimension. Notice that the implementation of Cannon's algorithm that uses matmul works correctly regardless of the number of levels of tiles in the hierarchy.

```
function C = summa (A, B, C)
  for k=1:m
     T1 = repmat(A{:, k}, 1, m);
     T2 = repmat(B{k, :}, m, 1);
     C = C + T1 * T2;
  end
```

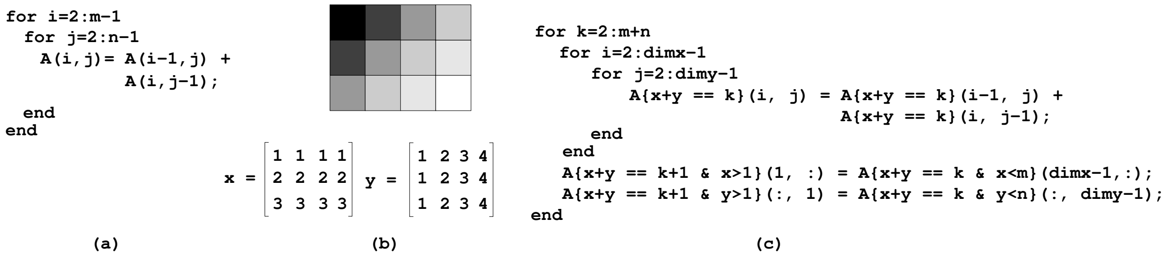
Figure 10. SUMMA Matrix Multiplication using HTAs.

3.1.3 Repmat

Another important type of communication is replication (repmat) which appears in the SUMMA Matrix algorithm [13] shown in Figure 10. This algorithm is based on the outer product version of the matrix multiplication. In SUMMA, the result C of the multiplication of the matrices A and B is computed as the addition of C with the outer product of column k of A and row k of B, for each possible value of k. In our implementation, matrices A and B are tiled and distributed one tile per processor across a two-dimensional processor mesh. The column of tiles A{:,k} is replicated on all columns of processors and the row of tiles B{k,:} is replicated along all rows of processors. This replication is achieved with an overloaded version of the repmat vector operator.

3.1.4 Logical Indexing

A more complex pattern of communication appears in parallel wavefront computations. This type of computations results from the parallelization of codes where the value of an element depends on the value or values of neighbors 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. The processors compute local data before sending them to the processors containing the dependent data. Wavefront computations can also be parallelized in a tiled fashion, and for that we used logical indexing. Figure 11-(a) shows a Fortran code with a 2D wavefront computation. The tiled HTA version is shown in Figure 11-(c), where logical indexing is used to determine the tiles that can operate in each iteration of the k loop. Those tiles where the condition (x+y == k) is true will locally compute the 2D wavefront computation. A pictorial view of how the computation advances across tiles is shown in Figure 11-(b), where the values of the x and y matrices are also shown. In the Figure, A is a $m \times n$ HTA, distributed on a $m \times 1$ processor mesh, so that rows of tiles are mapped to the same processor. The last two statements in Figure 11-(c) copy the last row and column of tiles that finished the computation in iteration k to the first row and column of the tiles that are going to start the computation in the iteration $k + 1$. A parallel wavefront similar to the one shown in Figure 11 appears in the LU NAS code.



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

  end
end
```

$$x = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} \quad y = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

```
for k=2:m+n
   for i=2:dimx-1
      for j=2:dimy-1
         A{x+y == k}(i, j) = A{x+y == k}(i-1, j) +
                             A{x+y == k}(i, j-1);
      end
   end
   A{x+y == k+1 & x>1}(1, :) = A{x+y == k & x<m}(dimx-1,:);
   A{x+y == k+1 & y>1}(:, 1) = A{x+y == k & y<n}(:, dimy-1);
end
```

Figure 11. 2-D wavefront computation.(a) Fortran code. (b) Pictorial view. (c) HTA code using logical indexing.

3.1.5 Redistribution

A different pattern of communication appears in a type of scientific applications that use Adaptive Mesh Refinement, where a portion of the mesh needs to be re-distributed across the processors. Redistribution can be done by selecting the section of the array underlying the HTA and specifying a new distribution: `A = hta(H(x1:x2, y1:y2), {`*partition vectors*`}, [m n])`, where HTA `H` contains the region `(x1:x2, y1:y2)`, that we want to redistribute. The region of interest is selected first, then a standard non-partitioned matrix is generated. This matrix is, in turn, partitioned using the appropriate partition vectors, in order to distribute the resulting tiles on a $m \times n$ processor mesh.

3.2 Global Computations

The simplest form of global computation is achieved by operating in parallel on a set of tiles from an HTA distributed across a parallel machine. This can be accomplished with `parHTA(@func, H)` that applies the function `func`. In MATLAB `@func` represents a pointer to function `func`. Many global computations take the form of reduction operations where the operator can be `sum`, `max`, or a user defined function. These operations can be expressed in HTA programs with the `reduceHTA` method. Figure 12 shows a matrix-vector multiplication where the `reduceHTA` method is applied. `A` is an HTA containing the matrix `MX` which is distributed across $m \times n$ processors and tiled as specified by `partition_A`. `B` is a two-dimensional HTA obtained by replicating the hta `V` which contains the vector `VX` to multiply. The HTA `V` is replicated `m` times as specified by the operator `repmat(V,m,1)`. Since `V` is a distributed HTA replication takes place along the vertical dimension of the processor mesh. The matrix-vector multiplication `A * B` takes place locally and each processor multiplies its portion of the matrix `A` by its portion of the vector in `B`. Notice that the row vector `B` has been first transposed within each processor by `parHTA(@transpose,B)` into a column. After the multiplication, a reduction along the rows (the second dimension as specified in the 3rd parameter of `reduceHTA`) will generate the resulting HTA `C` which consists of a column vector distributed across the m rows of our $m \times n$ mesh and replicated along its n columns of processors. This vector is replicated because the `reduceHTA` operator was invoked with the last parameter set to `true`, which indicates that it is an all-to-all reduction. The core computation of NAS `CG` benchmark is a sparse matrix-vector multiplication. Notice that the code is highly simplified by the overloading of array operators so that they also apply to sparse arrays.

4. Two Implementations

We have developed implementations of the HTA class for two sequential languages: MATLAB and C++. We have also implemented an extension of the X10 parallel language developed by IBM in which HTAs replace distributed arrays. We implemented HTAs as a class using the OO capabilities of each of these two languages. Although a single implementation could be used across languages, we decided to develop a different implementation for each one of these languages mainly because our first implementation, developed for MATLAB , demonstrated that, for many computations it is difficult to obtain reasonable absolute performance. To avoid the inefficiencies of MATLAB , we reimplemented the library in C++ and X10. To obtain the syntax of HTA shown in the previous sections, we applied the operator overloading capabilities of MATLAB . So far, we have not taken advantage of operator overloading in our C++ implementation and, as a result, the current notation is somewhat verbose, but this will be fixed. Below, we briefly discuss the MATLAB and C++ implementations.

```
A = hta(MX, {partition_A}, [m n]);
V = hta(VX, {partition_B}, [m n]);
B = repmat(V, m, 1)
B = parHTA(@tranpose, B)
C = reduceHTA('sum', A * B, 2, true);
```

Figure 12. HTA code for matrix-vector multiplication

4.1 MATLAB

The purpose of our first implementation on MATLAB was to demonstrate that a conventional sequential language could be easily extended for parallel computation using HTAs. MATLAB was a natural choice for this experiment because of its array syntax and OO capabilities. We found that MATLAB 's syntax for cell array accesses, generalized with triplet notation and extended to allow operations between components, was convenient to represent HTA accesses and therefore we adopted it. HTAs were implemented as a MATLAB toolbox programmed in both C and MATLAB with invocations to MPI primitives. The MATLAB toolbox mechanism proved adequate to implement with reasonable efficiency and natural syntax all needed HTA operations except for the `forall` array operation. However, we were able to develop an elegant implementation of all the codes we studied without `forall`.

Our first MATLAB implementation of HTAs followed the client/server model in which the main thread is executed on a workstation and HTAs are stored and manipulated in a distributed system that operates as a co-processor. Although this approach facilitates program understanding, it requires too much communication between the workstation and the processors in the background parallel machine. We decided to change the implementation to follow a SPMD execution model although the programmer could still think in terms of the client-server model to understand the functional behavior of the program (but, of course, not to analyze performance). This was achieved by executing the program on all processors and replicating on each processor scalar variables, arrays, and non-distributed HTAs. All processors redundantly execute the compu-

Nprocs	EP (CLASS C)		FT (CLASS B)		CG (CLASS C)		MG (CLASS B)		LU (CLASS B)	
	Fortran+ MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA
1	901.6	3556.9	136.8	657.4	3606.9	3812.0	26.9	828.0	15.7	245.1
4	273.1	888.8	109.1	274.0	362.0	1750.9	17.0	273.8	6.3	60.5
8	136.3	447.0	65.5	159.3	123.4	823.6	9.6	151.3	2.9	29.9
16	68.6	224.8	37.2	87.2	89.5	375.2	4.8	87.0	1.2	16.0
32	34.7	112.0	20.7	42.9	48.4	250.3	3.3	54.9	1.1	9.8
64	17.1	56.7	10.4	24.0	44.5	148.0	1.6	50.4	1.3	7.1
128	8.5	29.1	5.9	15.6	30.8	123.0	1.4	38.5	1.6	N/A

Table 1. Execution times in seconds for some of the applications in the NAS benchmarks for Fortran+MPI versus MATLAB +HTA. The execution time for 1 processor corresponds to the serial application in Fortran or MATLAB , without MPI or HTAs.

tation not involving distributed HTA operations. Since all data are replicated, the behavior in each processor is exactly the same as what would be the behavior of the client except that no communication is necessary to use data from the main thread in operations on distributed HTAs. On invocation of a method on a distributed HTA, each processor applies the corresponding operation to the tiles of the HTA it owns.

The incorporation of HTAs in MATLAB produced an explicitly parallel programming extension of MATLAB that integrates seamlessly with the language. Most other parallel MATLAB extensions either make use of extraneous primitives (MultiMATLAB [24]) or do not allow explicit parallel programming (Matlab*P [17]). Also, the incorporation of HTA gives MATLAB a mechanism to access and operate on tiles much more powerful than that provided by their native `cell arrays`. The main disadvantage of the implementation is that the immense overhead of the interpreted MATLAB limits the efficiency of many applications. The three main sources of this overhead are:

- *Excessive creation of temporary variables.* MATLAB creates temporaries to hold the partial results of expression, which significantly slows down the programs.
- *Frequent replication of data.* MATLAB passes parameters by value and assignment statements replicate the data, and
- *Interpretation of instructions.* The overhead resulting from the interpretation of instructions is more pronounced when the computation relies mainly on scalar operations.

Table 1 presents the execution time for Fortran+MPI and our MATLAB +HTA implementations of most of the NAS benchmarks. The table shows the execution times in seconds when the applications execute on a cluster of up-to 128 processors. Each processor is a 3.2 GHz Intel Xeon connected through a Gigabit Ethernet. For the NAS benchmarks we used the version 3.1, and compiled them with the INTEL ifort compiler, version 8.1, and flag -03. For MATLAB we used the version 7.0.1 (R14). Finally, for MPI we used MPI-LAM [6].

The execution time for 1 processor corresponds to the serial execution of the pure Fortran or MATLAB code without MPI or HTAs. Results in Table 1 correspond to the class C input for EP and CG, and class B for MG, FT and LU.

As can be seen in the table, in the case of EP and FT the parallel MATLAB code takes advantage of parallelism leading to execution times that are of the same magnitude as those of the Fortran+MPI code. In the case of CG our parallel MATLAB does reasonably well, although not as well as the Fortran+MPI version that obtains super-linear speedups when the number of processors is 64 or smaller. However, for MG and LU the performance of the sequential MATLAB implementation was slow and, in the case of MG, the parallel MATLAB does not improve upon the serial Fortran version. Similarly, for BT (not shown) the serial MATLAB version runs so slow that, even the parallel version is not comparable with its sequential Fotran counterpart. Overall, for EP, FT and CG where the sequential MATLAB version runs 1 to 5 times slower than the Fortran version, the parallel MATLAB implementation does reasonably well improving upon the serial Fortran version. In these cases, it could be said that parallelism at least compensates for the interpretation overhead. For 128 processors the parallel MATLAB obtains speedups of 30.9, 8.8 and 29.3 over the sequential Fortran counterpart for EP, FT and CG, respectively.

4.2 C++

In the C++ implementation, HTAs are represented as composite objects with methods to operate on both distributed and non-distributed HTAs. As in the case of MATLAB , MPI is used for communication and, while the programming model is single threaded, HTA C++ programs execute in SPMD form. To facilitate programming, our C++ implementation enforces an allocation/deallocation policy through reference counting as follows: (1) HTAs are allocated through factory methods on the heap. The methods return a handle which is assigned to a (stack allocated) variable. (2) All accesses to the HTA occur through this handle, which itself is small in size and typically passed by value across procedure boundaries. (3) Once all handles to an HTA disappear from the stack, the HTA and its related structures are automatically deleted from memory. This design permits sharing of sub-trees among HTAs and also precludes deallocation errors. Moreover, the temporary arrays that are for instance created during the partial evaluation of expressions, are handled through this mechanism and deleted automatically as early as possible.

Performance is one of the main goals of our C++ implementation. Methods were optimized and whenever possible specialized for specific cases. Also, the user is given control over the memory layout of non-distributed HTAs. In MATLAB the layout was in the hands of the system and the user had no way of influencing it. Finally, to enable efficient access to scalar components of HTAs, the implementation was organized to guarantee that hot methods were inlined. This last strategy enabled the codes written using the library to have performance similar to that of traditional (non-HTAs) implementations. For example, the code in Figure 13 represents the multiplication of two two-dimensional arrays recursively tiled. The code is similar to the MATLAB code shown Figure 8.

The code in Figure 13 shows the declaration of the HTAs `A`, `B`, and `C`. The function `alloc` is the factory method that creates the HTAs. It takes as input the complete tiling information for each HTA, number of tiles in each dimension `(xtiles,ytiles)`, tile size `(tile_size_x, tile_size_y)`, and memory layout (`ROW`, `COLUMN`, or `TILE`). The function `mult` is recursive. When the input

```
typedef Tuple<2> T;

HTA<double, 2, 1> A, B,C;
A =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
              T(tile_sz_x,tile_sz_y)),ROW);
B =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
              T(tile_sz_x,tile_sz_y)),ROW);
C =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
              T(tile_sz_x,tile_sz_y)),ROW);

template <int LEVEL> void mult(
                  HTA<double, 2, LEVEL> A,
                  HTA<double, 2, LEVEL> B,
                  HTA<double, 2, LEVEL> C) {
  int M = A.shape()[0].size();
  int N = B.shape()[0].size();
  int Q = B.shape()[1].size();
  for (int i = 0; i< M; i++) {
     for (int k = 0; k < N; k++) {
        for (int j = 0; j< Q; j++) {
           mult (A[T(i,k)], B[T(k,j)], C[T(i,j)]);
  }}}
}

void mult(double& A,double& B,double& C)
{
  C += A * B;
}
```

Figure 13. Recursive matrix multiplication in C++ using HTAs

```
template <> void mult(
           HTA<double, 2, 0> A,
           HTA<double, 2, 0> B,
           HTA<double, 2, 0> C){
  int M = A.shape()[0].size();
  int N = B.shape()[0].size();
  int Q = B.shape()[1].size();
  for (int i = 0; i< M; i++) {
     for (int k = 0; k < N; k++) {
        for (int j = 0; j< Q; j++) {
           C[T(i,j)]+=A[T(i,k)]*B[T(k,j)];
  }}}
}
```

Figure 14. Specialization of `mult` for Leaf HTAs

HTAs have 1 or more levels the general `mult` function is called. When the recursion reaches the scalars finally, the function `mult` for scalars is called.

It is possible to specialize `mult` and terminate the recursion at a different level. For example in Figure 14, the recursion ends at level 0. At this point, an optimized library generated code can also be used to perform the matrix-matrix multiplication. For example, in Figure 15 the mini-MMM code generated by ATLAS [25] is used. The mini-MMM code is optimized for matrix-matrix multiplication of smaller matrices that fits into the cache. It benefits from optimizations like register-level tiling, unrolling and prefetching. In all the figures, `T(i, j)` represents the subscript pair `(i, j)`. The code of Figure 13- 15 can be tuned for practically any memory hierarchy configuration. This can be accomplished with minimal modifications to the source code, by changing the number of levels and sizes of the tiles in the HTA constructor.

```
template <> void mult(
           HTA<double, 2, 0> A,
           HTA<double, 2, 0> B,
           HTA<double, 2, 0> C){
    ATL_Mini_MMM(A, B, C);
}
```

Figure 15. Specialization with a call to a wrapper function that in turn calls the mini-MMM code generated by ATLAS.

Approach	Implementation		Address Space		Control	
	Language	Library	Global	Local	MT	ST
CAF	√		√		√	
GAS		√	√		√	
HPF	√		√			√
HTA		√	√			√
MPI/PVM		√		√	√	
POET		√	√		√	
POOMA		√	√			√
Titanium	√		√		√	
UPC	√		√		√	
X10	√		√		√	
ZPL	√		√		√	

Table 3. Characterization of parallel programming infrastructures

To give an indication of the overhead of our current C++ implementation, we show in Table 2 the performance results in MFLOPs for matrix-matrix multiplication (MMM). Results are shown for different matrix sizes and six different versions: a naïve implementation of MMM with 3 nested loops (`Naïve 3 loops`), a tiled version with 6 loops (`Tiled 6 loops`), our HTA matmul implementation in Figure 13(`HTA naïve`), an HTA matmul implementation where the MMM code for level 0 has been implemented using the mini-MMM code generated by ATLAS [25] (`HTA+ATLAS`), `ATLAS` and the `INTEL MKL` library [1]. For ATLAS we used the MMM code with the parameter values that ATLAS found to be optimal for tile size, and register blocking parameters, among others. Notice that SSE multimedia extensions were not used for `HTA+ATLAS`. For the tiled implementations (all except `Naïve 3 loops`) we used a square tile of 36×36, which is the value that we found to be the optimal for the machine where we ran the experiments (an INTEL Pentium 4 with 3.0 Gz and 8KB in L1). For `Naïve 3 loops` and `Tiled 6 loops` we show results using the gcc compiler, version 3.2.3. For the HTA implementations we used g++, version 3.2.3, since our HTA implementation has been done in C++. For ATLAS we used the version 3.6.0 and for the INTEL MKL library we used the version 8.0. Notice that INTEL MKL runs faster than the others because it uses INTEL SSE2 vector extensions, while all the other versions use scalar code. As can be seen by comparing the HTA+ATLAS with the ATLAS column, the overhead introduced in our current implementation by one level of HTA is between 8 and 13.5%.

5. HTAs and Other Parallel Programming Infrastructures

In this section, we compare the HTA approach with some of the other parallel programming infrastructures. Table 3 presents a summary of the main characteristics of the programming infrastructures discussed in this section. The first column classifies the programming infrastructure according to the type of implementation. (1) libraries containing operations that represent communication and

Matrix Size	Naïve 3 loops	Tiled 6 loops	HTA naïve	HTA+ATLAS	ATLAS	Intel MKL(1)
504	161	657	675	2069	2387	3624
1008	150	649	679	2192	2384	3762
2016	133	632	675	2216	2492	3821
3024	135	644	668	2245	2509	3716
4032	36	588	613	2217	2519	3752

Table 2. Performance in MFLOPS for different versions of matrix-matrix multiplication. (1) MKL uses SSE2 vector extension.

data sharing on programs where the rest of the operations are represented in conventional, sequential constructs and (2) programming language constructs or directives designed to represent parallelism implicitly or explicitly. The second column classifies the infrastructures according to the address space seen by each component of the parallel program. Except for the message-passing library approach, where a thread of execution is only allowed to reference data located in the node where it is running (local view), all other programming models allow the threads to access data located in any node (global view). The third column distinguishes between those approaches where the operations of each individual thread must be specified separately (multiple-threaded or MT) and those which provide a single-threaded (ST) view of the computation. We now compare the library and language approaches in three separate subsections: discussing the HTA approach, other library approaches, and the language extension approach.

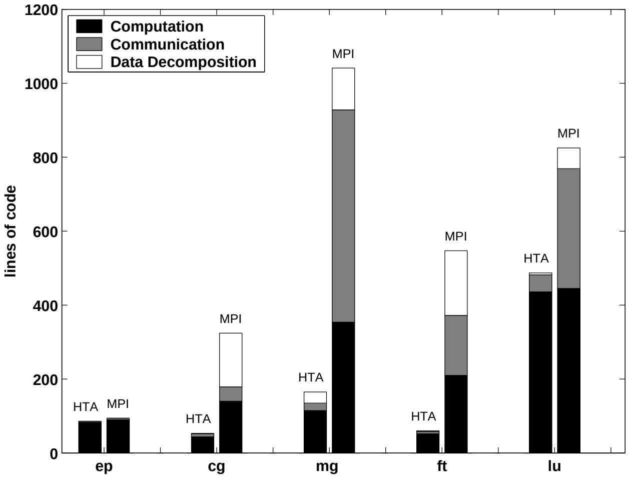


Figure 16. Linecount of key sections of HTA and MPI programs.

5.1 The HTA library

As we have seen in Section 4, the HTA class can be integrated in a very natural way in different languages thanks to operator overloading and the polymorphic features of current OO languages. Thus, the resulting programs tend to be more readable than those based on conventional libraries. As mentioned in the introduction, the most important characteristic that distinguishes HTAs from all other approaches is the consideration of the tile and its possible hierarchical decomposition as first-class concepts. This makes the HTAs ideal to design and write programs that can be naturally expressed in terms of blocks (e.g.. several matrix multiplication algorithms -Cannon [7], Summa [13]-, solvers such as LU, etc.) or which can be solved recursively (e.g., FT). Such blocks can be used to achieve parallelism or data locality or both, possibly using several levels of tiling for different purposes. With HTAs it is easy to adjust the point where recursive computations end and the iterative solutions start by changing the number of levels of tiling.

As shown in Table 3, HTA and POOMA [23] are the only library-based approaches that provide a global view of the data and follow a single threaded programming approach. This combination helps programmers' productivity in at least two ways. First, programmers can use familiar sequential programming languages, and reuse sequential modules, perhaps with small changes. Therefore, programmers can write parallel programs practically in the same way they write sequential programs. Second, the single-threaded semantics of HTAs eases the transition from sequential to parallel because programmers need not be concerned with the program's behavior on a per processor basis, deadlocks, race conditions, etc., since parallelism and synchronization are implicit. The single-threaded property also improves readability. Furthermore, in the case of HTA, flattening enables gradual migration of sequential applications to parallel form. This was the approach we followed in our translation of the NAS benchmarks from sequential MATLAB to the HTA-based parallel version.

A good indication of the benefit of the single-threaded form is obtained by comparing the number of lines of code of the HTA with those of MPI programs. Although the number of lines of code is not the best metric to measure ease of programming, it can give a rough estimate of program complexity. The plot in Figure 16 shows the lines of code for HTA and MPI codes. Since MATLAB language does not have declarations, we ignored those lines from the FORTRAN/MPI codes as well. Each bar shows the lines of code for the computation, communication and data decomposition sections of the codes. The difference in the number of lines of computation is not relevant to our discussion here since it is due to the characteristics of the MATLAB language, especially the availability of vector operations. However, the other two numbers are good indications of ease of programming and clearly show the advantages of HTAs. Thus, the lines of code for communication are significantly lower in HTA programs. The reason is that 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. HTAs are partitioned and distributed using the single HTA constructor, while MPI programs need to compute the limits of data owned by each processor, neighbors of a given processor, active set of processors in a given step of the program, etc.

A downside of the single-threaded approach is that asynchronous overlap is not easy to express explicitly. However, much of this overlap can be achieved automatically with the appropriate implementation.

A valuable property of a programming approach is the ability to convey to the programmers the cost of the execution of their code. This is particularly true in the case of parallel environments, where communication costs can easily dominate the execution time. HTAs are faithful to this idea: the statements that require data communication are clearly identified in the code either because of the usage of different indexes in the tiles, or because of the invocation of functions that involve data communication (transposition, circular shift, etc.).

HTAs have also drawbacks. For example, just as the other global-view approaches [12], they only allow limited forms of task-parallelism. Other limitations are due to the implementation. For example, our current implementation as a library forces to use dynamic analysis techniques to determine the communication patterns required when data is to be shuffled among processors. A compiler could calculate statically those patterns when they are regular enough, and generate a code with less overhead.

5.2 Other Library Approaches

The most popular parallel programming approach for distributed-memory systems is the use of a message passing library such as MPI [15] or PVM [14]. In this approach, the programmer has a local view of the data structures and must write programs that execute in a SPMD fashion. The communication and computation statements can be interleaved in an unstructured manner, potentially leading to programs that are difficult to understand and maintain.

An improvement over this approach is the usage of libraries that, while requiring SPMD programming, provide the user with a global view of the data structures. This is the case of the Global Arrays library [20] or the POET framework [4]. However, the SPMD programming style and the requirement of explicit synchronization complicates programming.

Other libraries like POOMA [23] integrate their classes in a host OO language, and exploit operator overloading and polymorphism in order to provide a global view of the data and a single-threaded view of the computation, as our HTA library. However, POOMA differs in fundamental ways from our approach. For example, while POOMA's arrays can be distributed in tiles, the library provides no easy means to explicitly refer to those tiles. Also, hierarchical decomposition is not natural to POOMA's arrays, while it is a defining property of HTAs.

5.3 Language/Compiler Based Approaches

Several infrastructures are based on new languages with constructs to control concurrency and distribution. As we can see in Table 3, all the language-based approaches provide a global view of the data, but they can be classified in two groups according to their view of the control flow: The multiple-threaded languages, like Co-Array Fortran (CAF) [21], Titanium [27],UPC [8] or X10 [11]; and single threaded form like High Performance Fortran HPF [16, 18] and ZPL [9]. All these approaches (except HPF) share a common drawback: they force programmers to rewrite their applications in parallel from scratch, an effort that can be ameliorated by providing interfaces with codes and libraries written in other languages.

5.3.1 Multiple-threaded Languages

The control model of many language-based infrastructures is explicitly parallel SPMD in which programmers are responsible for managing data distribution and low-level synchronization. An advantage of these languages is that they are much more suitable than the single-threaded counterparts for expressing task parallelism.

Another common characteristic of these languages is that they provide a Partitioned Global Address Space (PGAS), in which any thread of execution can create objects that can be accessed by other threads. And each thread and piece of data is associated with exactly one node of execution. They typically provide also constructs to distinguish between remote and local accesses. This helps programmers reason about the cost of their codes.

5.3.2 Single-Threaded Languages

In single-threaded languages, communication and synchronization are no longer responsibility of the programmer, but of the compiler. Programs written in this model tend to be shorter and easier to understand and maintain than those expressed in local view languages, which increases programmers' productivity. The downsides of these languages are their limited ability to express irregular parallelism and the responsibility they put on compiler technology, which may not be developed enough to generate efficient codes in some situations.

Different strategies have been studied to provide parallel codes with a global view of the algorithms to execute. For example, High Performance Fortran (HPF) [16, 18] annotates sequential Fortran codes with directives that specify array distribution, loop scheduling, etc. These directives are optional, and there is little information about how the compiler will translate them. The lack of a clear performance model makes it difficult for programmers to reason about an algorithm's performance.[19].

Another approach is design a language from scratch, which is the case of ZPL [9]. This language is designed in order to minimize the effort of the compiler. Its syntax allows to identify the operations that generate communication and their qualitative cost in a similar way to our HTAs.

6. Conclusions

In this paper we have introduced Hierarchically Tiled Arrays (HTAs). Our experience with the implementation of the NAS benchmarks and a few kernels using this new data type indicates that HTAs are an effective device for the development of high performance programs that are readable, easy to develop and maintain. During this study we determined that well-known array operations can be overloaded to represent communication and parallel computation and that, at least for the NAS benchmarks and the kernels we considered, are sufficient to represent efficient implementation of parallel algorithms and algorithms with a high degree of locality. We expect that the study reported in this paper will lead to useful portable libraries and provide insights useful for the further development of vector constructs and vector languages. This last issue is particularly important since vector operations are a powerful mechanism to express parallelism in a structured manner for many classes of algorithms.

Acknowledgments

This work was supported by the National Science Foundation (NGS program) under Grant No. 0103610. Basilio Fraguela was partially supported by the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2004-07797-C02-02), and the Galician Government (Project PGIDIT03-TIC10502PR).

References

[1] Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm.

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

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

[4] R. C. Armstrong and A. Cheung. POET (Parallel Object-oriented Environment and Toolkit) and Frameworks for Scientific Distributed Computing. In *Proc. of 30th Hawaii International Conference on System Sciences (HICSS 1997)*, pages 54–63, Maui, Hawai, 1997.

[5] G. H. Barnes, R. M. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes. The ILLIAC IV Computer. *IEEE Trans.*, 8(17):746–757, 1968.

[6] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

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

[8] 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.

[9] 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.

[10] B. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortrana Fortran Language Extension for Distributed Memory Multiprocessors. *Languages, Compilers and Run-time Environments for Distributed Memory Machines*, pages 39–62, 1992.

[11] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Procs. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Onward! Track*, Oct 2005.

[12] S. J. Deitz. Renewed Hope for Data Parallelism: Unintegrated Support for Task Parallelism in ZPL. Technical Report UW-CSE-03-12-04, University of Washington, Dec 2003.

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

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

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

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

[17] 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.

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

[19] T. A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1997.

[20] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable shared-memory programming model for distributed memory computers. In *Supercomputing '94: Proc. of the 1994 Conf. on Supercomputing*, pages 340–ff., Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

[22] D. Pham and et al. The Design and Implementation of a First-generation Cell Processor. In *Procs. of the IEEE Solid-State Circuits Symposium*, February 2005.

[23] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[24] 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.

[25] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[26] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *PLDI*, pages 30–44. ACM Press, 1991.

[27] 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.