# A Parallel Numerical Solver Using Hierarchically Tiled Arrays

James C. Brodman[1], G. Carl Evans[1], Murat Manguoglu[2], Ahmed Sameh[2], María J. Garzarán[1], and David Padua[1]

[1] University of Illinois at Urbana-Champaign, Dept. of Computer Science
{brodman2,gcevans,garzaran,padua}@illinois.edu
[2] Purdue University, Dept. of Computer Science
{mmanguog,sameh}@cs.purdue.edu

**Abstract.** Solving linear systems is an important problem for scientific computing. Exploiting parallelism is essential for solving complex systems, and this traditionally involves writing parallel algorithms on top of a library such as MPI. The SPIKE family of algorithms is one well-known example of a parallel solver for linear systems.

The Hierarchically Tiled Array data type extends traditional data-parallel array operations with explicit tiling and allows programmers to directly manipulate tiles. The tiles of the HTA data type map naturally to the block nature of many numeric computations, including the SPIKE family of algorithms. The higher level of abstraction of the HTA enables the same program to be portable across different platforms. Current implementations target both shared-memory and distributed-memory models.

In this paper we present a proof-of-concept for portable linear solvers. We implement two algorithms from the SPIKE family using the HTA library. We show that our implementations of SPIKE exploit the abstractions provided by the HTA to produce a compact, clean code that can run on both shared-memory and distributed-memory models without modification. We discuss how we map the algorithms to HTA programs as well as examine their performance. We compare the performance of our HTA codes to comparable codes written in MPI as well as current state-of-the-art linear algebra routines.

## 1 Introduction

Computer simulation has become an important tool for scientists and engineers to predict weather, forecast prices for financial markets, or test vehicle safety. Increasing the performance of these simulations is important to improve the accuracy of the prediction or to increase the number of tests that can be performed. One way to achieve this performance improvement is the parallelization of the kernels that lie at the core of many of these simulations and that solve systems of equations or perform signal transformations. Today many different types of computing platforms can be used to run these parallel codes, such as the new ubiquitous multicore, large clusters of machines where each node is typically a multicore, and the accelerators or clusters of accelerators like the Cell Processor or GPUs. However, the many available options for parallel execution

have increased the difficulty of the programmer's task as they usually must rewrite their computations with a different programming model for each different type of computing platform.

Programmer productivity can be improved with a programming model that produces one portable code that can target several different types of parallel platforms. We believe portable codes can be obtained by using high level abstractions that hide the details of the underlying architecture from the programmers and allow them to focus on the correct implementation of their algorithms. However, one does not want to raise the level of abstraction so high that programmers sacrifice control over performance. The Hierarchically Tiled Array (HTA) is a data type that uses abstractions to allow programmers to write portable numerical computations. HTA uses tiling as a high-level abstraction to facilitate the specification of the algorithms, while allowing the programmer to control the performance of their programs.

In this paper we show a proof-of-concept for high-performance computations that are portable across both shared-memory and message-passing. We present several versions of SPIKE, a parallel solver for linear banded systems, implemented using the HTA data type. We show that our implementations exploit the abstractions provided by the HTA to produce compact, clean code. Our experimental results show that the same code provides competitive performance when running on message-passing and shared-memory platforms.

The rest of this paper is organized as follows. Section 2 describes the SPIKE family of algorithms for solving banded systems of equations. Section 3 describes the Hierarchically Tiled Array data type used in our implementations. Section 4 describes how we actually implement several different SPIKE algorithms using HTAs. Section 5 presents the performance of our SPIKE implementations using both the shared-memory and distributed-memory runtimes and compares them to other libraries. Section 6 discusses related work and Section 7 summarizes our work.

## 2 SPIKE

Linear solvers are an important class of numerical computation. Many important problems are sparse. It is well known that the desired data structure to represent sparse systems influences the performance of solvers for this type of linear system. These computations do not use dense arrays but rather only store the elements of a matrix that may be non-zero. Such storage mechanisms reduce not only the memory footprint but can also reduce the amount of computation needed by only performing computation on relevant elements. The SPIKE family of algorithms [15] is one such parallel solver for banded linear systems of equations.

Consider a linear system of the form $Ax = f$, where $A$ is a banded matrix of order $n$ with bandwidth much less than $n$. One can partition the system into $p$ diagonal blocks. Given $p = 4$, the partitioned system is of the form,

$$A = \begin{bmatrix} A_1 & B_1 & & \\ C_2 & A_2 & B_2 & \\ & C_3 & A_3 & B_3 \\ & & C_4 & A_4 \end{bmatrix} \qquad f = \begin{bmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ \boldsymbol{f}_3 \\ \boldsymbol{f}_4 \end{bmatrix}$$

where each $A_i$ is a banded matrix of order $n/p$. The matrices $B_i$ and $C_i$ are of order $m$ where the bandwidth of the original matrix $A$ is $2m+1$. Only the $A$, $B$, and $C$ blocks need to be stored for this type of sparse matrix.

Let the block diagonal matrix $D = \mathrm{diag}(A_1, ..., A_4)$. If one were to left-multiply each side of the above by $D^{-1}$, one would obtain a system of the form:

$$S = \begin{bmatrix} I & V_1 & & \\ W_2 & I & V_2 & \\ & W_3 & I & V_3 \\ & & W_4 & I \end{bmatrix} \qquad g = \begin{bmatrix} \boldsymbol{g}_1 \\ \boldsymbol{g}_2 \\ \boldsymbol{g}_3 \\ \boldsymbol{g}_4 \end{bmatrix}$$

However, instead of computing $D^{-1}$, one can compute, as seen below, the blocks of $V$ and $W$, or, the *spikes* by solving a system of equations. The spikes have the same width, $m$, as the $B$ and $C$ tiles in the original system.

$$A_i \left[V_i, W_i\right] = \begin{bmatrix} 0 & C_i \\ . & 0 \\ . & . \\ 0 & . \\ B_i & 0 \end{bmatrix} \tag{1}$$

Solving the original system $Ax = f$ now consists of three steps.

1. Solve (1)
2. Solve $Dg = f$
3. Solve $Sx = g$

The solution of the system $Dg = f$ yields the modified RHS for the system in the third step. Notice that each blocks of $D$ are independent and thus can be computed in parallel. Solving the third system can be further reduced by solving the system $\hat{S}\hat{x} = \hat{g}$, which consists of the $m$ rows of $S$ directly above and below the boundaries between the $I$ tiles. The spikes, $f$, and $g$ can also be partitioned so that we have:

$$V_j = \begin{bmatrix} V_j^{(t)} \\ V_j' \\ V_j^{(b)} \end{bmatrix} \quad W_j = \begin{bmatrix} W_j^{(t)} \\ W_j' \\ W_j^{(b)} \end{bmatrix} \quad x_j = \begin{bmatrix} x_j^{(t)} \\ x_j' \\ x_j^{(b)} \end{bmatrix} \quad g_j = \begin{bmatrix} g_j^{(t)} \\ g_j' \\ g_j^{(b)} \end{bmatrix} \tag{2}$$

The reduced system thus takes the following form:

$$\begin{bmatrix}
I_m & 0 & V_1^{(t)} & & & & & & \\
0 & I_m & V_1^{(b)} & 0 & & & & & \\
0 & W_2^{(t)} & I_m & 0 & V_2^{(t)} & & & & \\
 & W_2^{(b)} & 0 & I_m & V_2^{(b)} & 0 & & & \\
 & & \ddots & \ddots & \ddots & \ddots & \ddots & & \\
 & & & 0 & W_{p-1}^{(t)} & I_m & 0 & V_{p-1}^{(t)} & \\
 & & & & W_{p-1}^{(b)} & 0 & I_m & V_{p-1}^{(b)} & 0 \\
 & & & & & 0 & W_p^{(t)} & I_m & 0 \\
 & & & & & & W_p^{(b)} & 0 & I_m
\end{bmatrix}
\begin{bmatrix} \mathrm{x}_1^{(t)} \\ \mathrm{x}_1^{(b)} \\ \mathrm{x}_2^{(t)} \\ \mathrm{x}_2^{(b)} \\ \vdots \\ \mathrm{x}_{p-1}^{(t)} \\ \mathrm{x}_{p-1}^{(b)} \\ \mathrm{x}_p^{(t)} \\ \mathrm{x}_p^{(b)} \end{bmatrix} =
\begin{bmatrix} \mathrm{g}_1^{(t)} \\ \mathrm{g}_1^{(b)} \\ \mathrm{g}_2^{(t)} \\ \mathrm{g}_2^{(b)} \\ \vdots \\ \mathrm{g}_{p-1}^{(t)} \\ \mathrm{g}_{p-1}^{(b)} \\ \mathrm{g}_p^{(t)} \\ \mathrm{g}_p^{(b)} \end{bmatrix}$$

Finally, once the solution to the reduced system has been directly computed sequentially, we will have the values of $x^{(b)}$s and $x^{(t)}$s. The rest of $x$ can then be computed as follows:

$$\begin{cases} x_1' = g_1' - V_1' x_2^{(t)}, \\ x_j' = g_j' - V_j' x_{j+1}^{(t)} - W_j' x_{j-1}^{(b)}, j = 2, ..., p-1, \\ x_p' = g_p' - W_p x_{p-1}^{(b)}. \end{cases} \tag{3}$$

Thus the SPIKE algorithm can be broken down into the following steps:

1. Factorize the diagonal blocks of $A$.
2. Compute the spikes using the factorization obtained in the previous step and compute the right hand side. Solve (1) and $Dg = f$.
3. Form and solve the reduced system.
4. Compute the rest of x.

### 2.1 SPIKE Variants

The original SPIKE algorithm explained above has many variants. These variants target systems of equations with certain properties in order to reduce the amount of computation performed. They also increase the amount of parallelism available during different stages of the algorithm. In this paper we focus on two variants that use a truncated scheme to solve the reduced system. The truncated scheme is useful for systems that are diagonally dominant. In diagonally dominant systems, the values in the spikes far from the diagonal are likely to be very close to zero and therefore contribute little to the solution. Consequently, the truncated scheme treats these values as zero and only computes the $m \times m$ portion of the spikes close to the diagonal, specifically, $V^{(b)}$ and

$W^{(t)}$. This is accomplished by either using the LU or UL factorization computed for the blocks of the diagonal.

The two variants we present are called `TU` and `TA`, and both implement the truncated scheme. LU factorization of $A_i$ is used to solve the bottom tips, $V_i^{(b)}$, of the spikes and the UL factorization of $A_i$ is used to solve for the top tips, $W_i^{(t)}$, of the spikes. The difference between `TU` and `TA` lays in the decomposition of the work. In the `TU` scheme, the original matrix is partitioned into as many blocks as there are processors. Figure 1 shows this partitioning for the case with 4 processors. In this figure $\hat{B}_i$ and $\hat{C}_i$ are $\begin{bmatrix}0 \ldots 0 \ B_i\end{bmatrix}^T$ and $\begin{bmatrix}C_i \ 0 \ldots 0\end{bmatrix}^T$ as in equation 1.

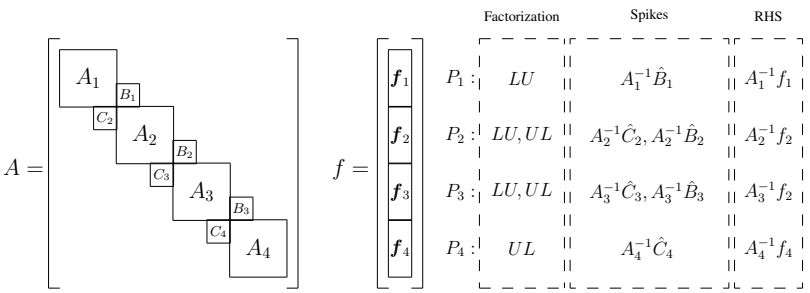



Fig. 1: Spike `TU` Partitioning

The `TA` scheme arises from the fact that the factorization step dominates execution time. `TA` is similar to `TU` with the exception that it partitions the matrix in a different fashion. Instead of each processor computing both LU and UL for a block since some blocks must compute two spikes, each processor now computes either LU or UL for a block but not both in order to compute a single spike. Note that this scheme partitions the matrix into fewer blocks than the `TU` scheme does, but results in better load balance for the computation of the spikes. Figure 2 shows this partitioning for 4 processors using $\hat{B}_i$ and $\hat{C}_i$ which are extended as above.

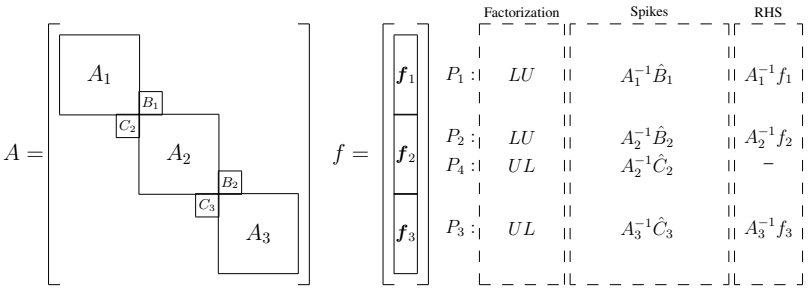



Fig. 2: Spike `TA` Partitioning

Both versions of the algorithm compute the $W^{(t)}$, $V^{(b)}$, and $g$ tips that are needed for the truncated reduced system, shown in Figure 3. This system will be block diagonal and has one less block than the original system. Thus when solving with $p$ processors `TU` will have $p-1$ blocks in the reduced system while `TA` will have $(p+2)/2-1$ blocks in the reduced system. Thus the `TU` version will have more parallelism than the `TA` version in this stage of the computation. Unlike the original SPIKE algorithm, the reduced system for truncated schemes can be solved in parallel via a direct scheme where each block has the following form:

$$\begin{bmatrix} I_m & V_j^{(b)} \\ W_{j+1}^{(t)} & I_m \end{bmatrix} \begin{bmatrix} x_j^{(b)} \\ x_{j+1}^{(t)} \end{bmatrix} = \begin{bmatrix} g_j^{(b)} \\ g_{j+1}^{(t)} \end{bmatrix} \tag{4}$$

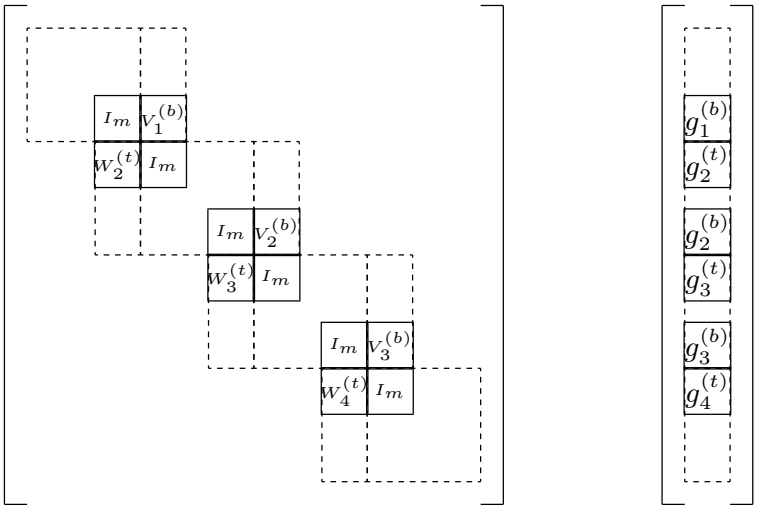



Fig. 3: Data sources for `TU` reduced with 4 blocks

Finally the solution to the original system is recovered by solving:

$$A_j x_j = f_j - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ B_j \end{bmatrix} x_{j+1}^{(t)} - \begin{bmatrix} C_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} x_{j-1}^{(b)} \tag{5}$$

This can be done in parallel with either the LU or UL factorization of $A_j$. Here again the `TU` version has more parallelism the the `TA` version.

## 3 Hierarchically Tiled Arrays

The Hierarchically Tiled Array [4, 9, 3], or HTA, data type extends earlier work on data parallel array languages with explicit tiling. An HTA object is a tiled array whose elements can be either scalars or tiles. HTAs can have several levels of tiling, allowing them to adapt to the hierarchical nature of modern machines. Figure 4 illustrates two examples of how HTAs can exploit hierarchical tiling. For example, tiles in the outermost level are distributed across the nodes in a cluster; then, the tile in each node can be further partitioned among the processors of the multicore node.

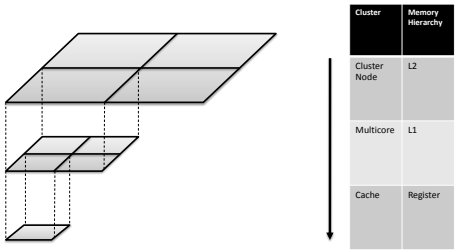



Fig. 4: Hierarchical Tiling

The HTA data type makes tiles first class objects that are explicitly referenced and extends traditional Fortran 90 style array operations to function on tiles. Figure 5 illustrates the ways in which HTAs can be indexed. HTAs permit indexing of both tiles and scalars. We use () to refer to tiles and [] to refer to elements. This way, A(0,0) refers to the top left tile of HTA A and A(0,1) [0,1] refers to the element [0,1] of the top right tile of HTA A. Also, HTAs support the triplet array notation in order to index multiple scalars and/or tiles, as shown in Figure 5 when accessing the two bottom tiles of A by using A(1, 0:1). Scalars can also be accessed in a flattened fashion that ignores the tiling structure of the HTA, as shown in the example when accessing the element A[0,3]. This flattened notation is useful for tasks such as initializing data.

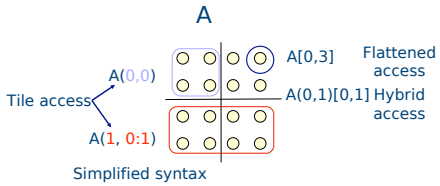



Fig. 5: HTA Indexing

HTAs provide several data parallel operators to programmers. One example is element-by-element operations such as adding or multiplying two arrays. HTAs also provide

support for operations such as scans, reductions, matrix multiplication, and several types of transpositions of both tiles and data. Communication of data between tiles is usually expressed through array assignments, but can also be expressed with special operations such as transpositions or reductions.

The HTA also provides a map operator that applies a programmer-specified function to the corresponding tiles of a set of HTAS. On a parallel platform these functions may be executed in parallel. This way, the `A.hmap(func1())` will invoke `func1()` on all the tiles of HTA A. If another HTA B is passed as an argument of `hmap`, then `func1()` will execute on the corresponding tiles of both HTAs, A and B.

HTA Programs thus consist of a sequence of data parallel operators applied to HTAs that are implicitly separated by a barrier. These programs appear sequential to the programmer as all parallelism is encapsulated inside the operators. Numbers and sizes of tiles are chosen both to control the granularity of parallelism and to enhance locality.

The HTA data type has been implemented as libraries for both C++ and MATLAB. The C++ library currently supports two platforms: distributed-memory built on top of MPI and shared-memory built on top of Intel's Threading Building Blocks. These multiple backends allows programmers to write one code using HTAs that can run on either multicores or clusters.

## 4 Implementing SPIKE with HTAs

An implementation of the SPIKE family of algorithms is available in the Intel Adaptive Spike-based Solver[1], or SpikePACK. It is implemented using MPI and Fortran. We choose to implement several SPIKE algorithms using HTAs for two reasons. First, writing SPIKE using HTAs would allow programmers to write one portable code that can be run on both shared-memory and distributed-memory target platforms. Second, the HTA notation allows for an elegant, clean implementation of the algorithms. An HTA SPIKE would more closely resemble the high-level mathematical expression of the algorithms than Fortran+MPI. Communication takes the form of simple array assignments between tiles.

We have implemented the `TU` and `TA` variants of SPIKE with HTAs. The tiles of the HTAs map to the blocks of the banded linear system. The bands of the system are stored inside the tiles using the banded storage format used by LAPACK. Since the code makes extensive use of LAPACK routines such as `DGBTRF` and `DGBTRS` to factorize and solve banded systems, we modified the HTA library to support column-major data layout due to the Fortran origins of these routines. The HTA library is written in C++ and originally only supported row-major layout.

The blocks of the bands, spikes, and reduced system are all represented as HTA objects. Each of these collections of blocks can be viewed as an array of tiles. The storage for the blocks of the band is overwritten to store the LU or UL factorizations of each block. The storage for the `B` and `C` blocks is likewise overwritten to contain the tips of the spikes. The number of partitions used by the algorithm for a given number of processors directly determines the tiling of the HTA objects. The algorithm is represented as a sequence of data parallel operations. The semantics state that each data parallel operation is followed by an implicit barrier. This allows the programmer to reason about

the algorithm sequentially as the parallelism is thus encapsulated inside of the data parallel operators. The data parallel operations often are represented as `hmap` operations. This is the mechanism through which we apply LAPACK kernels in parallel across all the tiles of an HTA. Our implementations also use the array operations provided by the HTA library to construct the reduced system. When coupled with HTA's first class tile objects, array operations enable programmers to write simple, compact statements that can communicate a range of data from one set of tiles to another. This contrasts with a Fortran+MPI approach where it is difficult to separate the algorithm from the implementation.

Porting the programs from one platform to another is accomplished by simply changing the header file for the library. In order to target MPI, one includes `htalib_mpi.h`. In order to target TBB, one includes `htalib_shmem.h`.

### 4.1 TU

Figure 6 presents the core of our implementation. We use a simplified notation to represent triplets. Recall that `TU` partitions the matrix into as many blocks as processors. The HTAs `LUA` and `ULA` initially are identical and contain the diagonal blocks of the system. The LU and UL factorizations of these blocks are performed in-place and in parallel by the `hmap` operators used in lines 3-4. The off-diagonal blocks, `B` and `C`, that will contain the spike tips are stored in the HTA `BC`. Each tile of this HTA contains space for both the "left" ($W^{(t)}$) and "right" ($V^{(b)}$) spikes associated with each block. The spike tips are computed in line 7 using the LU and UL factorizations computed previously. The whole right-hand side (RHS) stored in `g` for the system is then updated in line 10 using the LU factorization of the diagonal blocks.

The reduced system, shown in Figure 3, can be formed now that the spikes and updated RHS have been computed. Lines 13-16 make use of HTA array assignments to construct the reduced system by copying the spike tips into the appropriate sections of each block of the reduced system. The HTAs `REDUCED` and `BC` are indexed using () and [] operators and triplet notation. The first () is shorthand for selecting every tile of the HTA `REDUCED`. For the HTA `BC`, we select different ranges of tiles for each statement. The [] operator is used to index a range of elements inside of a tile. The RHS of the reduced system is formed similarly in lines 19-20. Note that the array assignments used to form the reduced system imply communication. Once the reduced system has been formed, it may be solved in parallel as its blocks are independent. This is accomplished by calls to the `hmap` operator on lines 23 and 25.

Having solved the reduced system, the RHS of the original system is updated in lines 28-33. This is accomplished by array assignments and another call to `hmap` that performs matrix-vector multiplications in parallel. Once the RHS has been updated with the values computed from the reduced system, the rest of the solution is obtained in line 36.

Our implementation of the `TU` scheme slightly deviates from the SpikePACK implantation of the algorithm in two ways. First, the first and last partitions need only compute LU or UL, respectively. The inner partitions must compute both LU and UL in order to compute the tips of the left and right spikes. The first and last partitions only have either a right or a left spike and do not need to compute both. However, we chose

to have the first and last partitions compute a fake spike in order to avoid special cases when computing the spikes. We compute both LU and UL for all partitions where as the SpikePACK only computes the LU for the first and the UL for the last as needed by the algorithm. Secondly the SpikePACK implementation uses a nonuniform distribution with larger partitions for the first and last partitions to balance the load since they are only computing one factorization. Since we compute two factorizations for every partition, our implementation uses a uniform size distribution.

```
1  ...
2  // factorize blocks of A
3  LUA.hmap(factorize_lua());
4  ULA.hmap(factorize_ula());

6  // calculate the spike tips W(t) and V(b) from Bs and Cs
7  BC.hmap(solve_bc(),LUA,ULA);

9  // update right hand side
10 g.hmap(solve_lua(),LUA);

12 // form the reduced system
13 REDUCED()[0:m-1,m:2*m-1] =
14         BC(0:num_blocks-2)[0:m-1,0:m-1];
15 REDUCED()[m:2*m-1,0:m-1] =
16         BC(1:num_blocks-1)[0:m-1,m:2*m-1];

18 // form the reduced system RHS
19 greduced()[0:m-1] = g(0:num_blocks-2)[blocksize-m:blocksize-1];
20 greduced()[m:2*m-1] = g(1:num_blocks-1)[0:m-1];

22 // factorize the reduced system
23 REDUCED.hmap(factorize());
24 // solve the reduced system
25 greduced.hmap(solve(),REDUCED);

27 // Update RHS with the values from the spikes as r = r - Bz - Cz
28 fv = r(0:num_blocks-2); fr_half = greduced()[0:m-1];
29 B.hmap(dgemv(),fv,fr_half);
30 r(0:num_blocks-2) = fv;
31 fw = r(1:num_blocks-1); fr_half = greduced()[m:2*m-1];
32 C.hmap(dgemv(),fw, fr_half);
33 r(1:num_blocks-1) = fw;

35 // Solve the updated system
36 r.hmap(solve_lua(),LUA);
37 ...
```

Fig. 6: HTA SPIKE TU

### 4.2 TA

The implementation of the `TA` variant is structurally similar to our implementation of `TU`. Figure 7 presents the core of our implementation of `TA`. The algorithm consists of array assignments and calls to the `hmap` operator. The main difference from `TU` is that each processor now computes either the LU or the UL factorization for a block

but not both. The TU variant partitions the matrix into one block per processor, and some processors must compute two spikes. TA has each processor compute only one spike. Consequently TA partitions the matrix into fewer blocks for a given number of processors than TU as shown in Figure 2. Whereas TU stored the diagonal blocks in the HTAs LUA and ULA, TA stores the appropriate blocks in the HTA DIAGS. Note that DIAGS can contain two copies of the same block of $A$ since the same block is needed to compute two different spikes for the inner blocks. An additional HTA, DIAG_MAP, is used to set flags that indicate whether each tile needs to perform the LU or the UL factorization for its block. This can be seen in line 3 for the factorization and line 7 for the computation of the spike tips. The HTA TOSOLVERHS is used to refer to part of DIAGS as that HTA can contain multiple factorizations for each block. TOSOLVERHS, seen on line 4, contains only one factorization for each block of the matrix and is used to update the right hand side on lines 9 and 35. This is also matched with a map that indicates the type of factorization contained in the tile. Forming and solving the reduced system proceeds almost identically to the implementation of TU. Note that there is less parallelism available in this phase of TA than in TU due to partitioning the system into fewer blocks.

```
1  ...
2  // factorize the A blocks
3  DIAGS.hmap(factorize_diag(),DIAG_MAP);
4  TOSOLVERHS = DIAGS(0:num_blocks-1);

6  // compute the spike tips from Bs and Cs
7  BC.hmap(solve_bc(),DIAG_MAP,DIAGS);
8  // generate modified right hand side
9  g.hmap(solve_rhs(),TOSOLVERHS_MAP,TOSOLVERHS);

11 // form the reduced system
12 REDUCED()[0:m-1,m:2*m-1] =
13         BC(0:num_blocks-2)[0:m-1,0:m-1];
14 REDUCED()[m:2*m-1,0:m-1] =
15         BC(num_blocks-1:2*num_blocks-3)[0:m-1,0:m-1];

17 // form the reduced system right hand side
18 greduced()[0:m-1] = g(0:num_blocks-2)[blocksize-m:blocksize-1];
19 greduced()[m:2*m-1] = g(1:num_blocks-1)[0:m-1];

21 // factorize the reduced system
22 REDUCED.hmap(factorize());
23 // solve the reduced system
24 greduced.hmap(solve(),REDUCED);

26 // Update RHS with the values from the spikes as r = r - Bz - Cz
27 fv = r(0:num_blocks-2); fr_half = greduced()[0:m-1];
28 B.hmap(dgemv(),fv,fr_half);
29 r(0:num_blocks-2) = fv;
30 fw = r(1:num_blocks-1); fr_half = greduced()[m:2*m-1];
31 C.hmap(dgemv(),fw,fr_half);
32 r(1:num_blocks-1) = fw;

34 // Solve the updated system using the LU and UL as needed
35 r.hmap(solve_rhs(),TOSOLVERHS_MAP,TOSOLVERHS);
36 ...
```

Fig. 7: HTA SPIKE TA

## 5 Experimental Results

In order to evaluate the performance of our HTA implementations of the two spike variants, we conducted several experiments. We compare the performance of our implementations to both the SPIKE implementations in the Intel®Adaptive Spike-Based Solver version 1.0 and the sequential banded solvers found in the Intel®Math Kernel Library version 10.2 Update 5. The numbers reported are speedups over the sequential MKL routines, `DGBTRF` and `DGBTRS`. All code was compiled with the Intel®compilers icc and ifort version 11.1 Update 6, and all MPI programs were run using mpich2. The shared-memory HTA library runs on TBB version 2.2 Update 3.

In all cases several different systems of equations were tested and the results were similar. We present one for each algorithm. Tests were run on a four socket 32-core system using Intel®Xeon®L7555 processors running at 1.86 GHz. The system has 64 gigabytes of memory installed and on a cluster at University of Massachusetts with 8 compute nodes each with two Intel®Xeon®X5550 processors running at 2.66 GHz connected with InfiniBand. In testing we experienced large variations in the execution time of all programs due to the use of a shared system. To control for this all tests were run 8 times and the minimum execution time is reported.

### 5.1 TU

We present the test for a matrix of order 1048576 with a bandwidth of 513 here. This size was chosen in order to partition the matrix into blocks of uniform size. Figures 8a and 8c plot the speedups over sequential MKL for `TU` running on HTAs for shared-memory run on the 32-core shared memory system, HTAs for distributed-memory, and the Intel SpikePACK run on both the shared memory system and the cluster.

We believe that our performance advantage comes from implementation differences. SpikePACK uses larger blocks for the first and last partitions to attempt to minimize any load imbalance when computing factorizations and the spikes. However, this creates imbalance when retrieving the solution to the whole system after the reduced system has been solved since the retrieval for the outer blocks will require more time than the retrieval for inner blocks. As the number of processors increases, the retrieval becomes a larger portion of the total execution, and this imbalance is magnified.

It is also important to note that the performance of the HTA codes on shared-memory is almost identical with both the mpi and tbb backend. While at first this result surprised us, it is indeed what we should expect. The amount of computation is large, so the overheads of each runtime system are minimal. The ideal tiling structure may differ from one platform to the next, but a given tiling ought to perform similarly on the same system regardless of the backend.

### 5.2 TA

We present the test for a matrix of order 1093950 with a bandwidth of 513 here. This size was again chosen to partition the matrix into blocks of uniform size. Recall that the `TA` scheme partitions the matrix into fewer blocks than the `TU` scheme for a given number of processors. `TU` assigns one block of the matrix per processor while `TA` assigns

one spike calculation per processor. The results of these tests are presented in Figures 8b and 8d which again shows speedup over sequential MKL for the three implementations. Each version tends to outperform `TU` and scales reasonably with increasing processors. However, SpikePACK begins to outperform the HTA implementations after 16 processors.

The performance difference seen in this case is due to the differences in the communication patterns between the HTA versions and the SpikePACK version. In the SpikePACK version of the algorithm, care is taken so that only one of the tips needs to be communicated to build the reduced system. This produces an irregular distribution of data. In cases where the number of partitions is small, distribution does not have a large impact but as the number of partitions grow the impact becomes more significant.

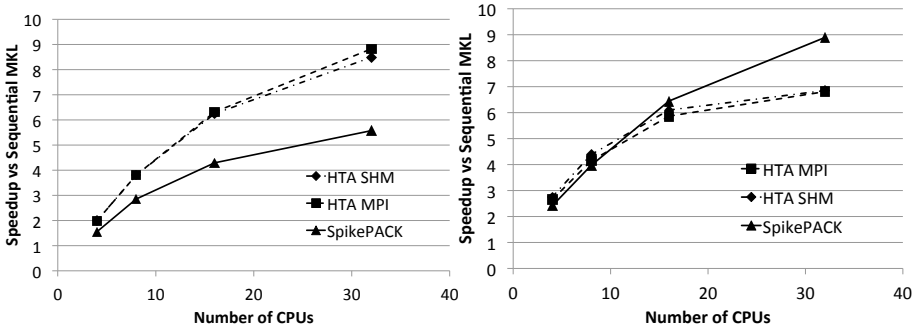
We believe that this behavior could implemented in the HTA versions of `TA` in two ways. First, the version of the library built on top of MPI provides support for user-defined distributions. These distributions could map the tiles of the spikes, RHS, and reduced system in such a way that minimizes communication between processors. The HTA library for shared-memory currently has no analog. This limitation is inherent in many libraries for shared-memory programming as they do not expose mechanisms to bind a thread to a particular core. The second way through which we could mimic SpikePACK's performance is through changing our implementation of the algorithm. By storing the blocks of the reduced system in a different order, we could more closely align the respective tiles of the spikes and RHS with the appropriate tiles of the reduced system. However, this complicates the implementation as the programmer becomes responsible for maintaining the mapping of the blocks of the reduced system to their locations in the HTA's tiling structure. We chose to initially focus on implementing a simple, elegant solution that closely maps to the algorithm.

## 6 Related Work

Implementing the SPIKE algorithms on top of the Hierarchically Tiled Array exploits both the portability and explicit tiling of the HTA programming model. Tiling has been extensively studied to improve performance of scientific and engineering codes [2, 11, 13, 17] for parallel execution [16] and as a mechanism to improve locality [17]. However, most programming languages do not provide any support for tiles. In languages such as C or Fortran, either the programmer needs to write the code to support tiled computations or the programmer must rely on the compiler to generate them.

Languages such as HPF [10, 12] or UPC [5] include support to specify how an array should be tiled and distributed among the processors, but the resulting tiles are only accessed directly by the compiler, and the programmer must use complex subscript expressions. Others like Co-Array Fortran [14] allow the programmer to refer to tiles and portions of them, but their co-arrays are subject to many limitations. Thus, the main difference of these languages with HTAs is that HTA Tiles are first class objects that are explicitly referenced, providing programmers with a mechanism for controlling locality, granularity, load balance, data distribution, as well as communication.

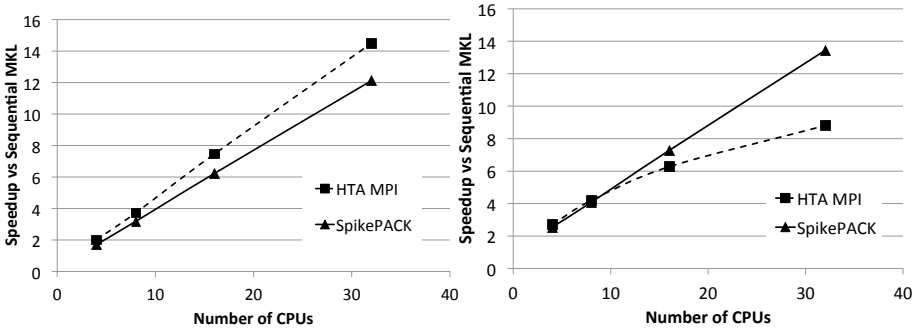
Sequoia [8] makes uses hierarchies of tasks to control locality and parallelism. Data is partitioned to create the parameters for the next level of tasks. In Sequoia, tasks




(a) TU Speedups




(b) TA Speedups




(c) Cluster TU Speedups




(d) Cluster TA Speedups

Fig. 8: Speedups over Sequential MKL

communicate by passing parameters to children tasks and by accepting return values from them. HTA on the other hand, is data centric so that tiling is associated with each object and parallel computation follows the tiling. This, combined with the array notation of HTAs, simplifies the notation when programming algorithms that use tiled objects. Furthermore, the HTA semantics does not require insulation of the operation on tiles and therefore subsumes that of Sequoia.

Many Partitioned Global Address Space, or PGAS, languages aim to provide support for writing a single program that can run on many different platforms. Examples of these languages include X10 [7], UPC [5], Chapel [6], and Titanium [18]. These languages exploit locality by using distribution constructs or directives as hints to the compiler on how to partition or map the "global" array to the different threads. However, programmers cannot directly access these tiles and can only use flat element indexes to access the data (which is similar to our flattened notation). The explicit tiles of HTA programs increase programmability because they represent better the abstraction that the programmer has of how data are distributed. Programming using flat indexes forces the programmer to recover the implicit tiling structure of the data when data communication is required.

## 7 Conclusions

In this paper we have shown through the implementation of two variants from the SPIKE family of algorithms that the Hierarchically Tiled Array data type facilitates portable parallel programming and increases productivity. Tiles facilitate the mapping of block algorithms to code and result in programs that can run without modifications on both shared-memory and distributed-memory models.

Our experimental results show that the performance of the same HTA code when running on both shared-memory and distributed-memory models achieve almost identical performance, and are competitive to the reference Intel library implemented on top of MPI. In addition, our codes show that the features provided by the HTA result in programs that are both clean and compact and closely resemble the algorithm description of the problem.

### Acknowledgments

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corporation and Microsoft Corporation.

## References

1. Intel adaptive spike-based solver, http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/
2. Abu-Sufah, W., Kuck, D.J., Lawrie, D.H.: On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. IEEE Trans. Comput. 30(5), 341–356 (1981)

3. Andrade, D., Fraguela, B.B., Brodman, J., Padua, D.: Task-parallel versus data-parallel library-based programming in multicore systems. Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0, 101–110 (2009)
4. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. pp. 48–57 (2006)
5. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences (1999)
6. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)
7. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: 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)
8. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Supercomputing '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. p. 83 (2006)
9. Guo, J., Bikshandi, G., Fraguela, B.B., Garzarán, M.J., Padua, D.: Programming with Tiles. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. pp. 111–122 (Feb 2008)
10. High Performance Fortran Forum: High Performance Fortran specification version 2.0 (January 1997)
11. Irigoin, F., Triolet, R.: Supernode Partitioning. In: POPL '88: Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 319–329 (1988)
12. Koelbel, C., Mehrotra, P.: An Overview of High Performance Fortran. SIGPLAN Fortran Forum 11(4), 9–16 (1992)
13. McKellar, A.C., E. G. Coffman, J.: Organizing Matrices and Matrix Operations for Paged Memory Systems. Communications of the ACM 12(3), 153–165 (1969)
14. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
15. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the spike algorithm. Parallel Computing 32(2), 177–194 (2006)
16. Ramanujam, J., Sadayappan, P.: Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. In: Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 111–120 (1991)
17. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proc. of the Conf. on Programming Language Design and Implementation. pp. 30–44 (1991)
18. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. In: Workshop on Java for High-Performance Network Computing (February 1998)