Programming with Tiles

Jia Guo, Ganesh Bikshandi[†], Basilio B. Fraguela[‡], María J. Garzarán, David Padua

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

[†]IBM, India
gbikshan@in.ibm.com

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

Abstract

The importance of tiles or blocks in scientific computing cannot be overstated. Many algorithms, both iterative and recursive, can be expressed naturally if tiles are represented explicitly. From the point of view of performance, tiling, either as a code or a data layout transformation, is one of the most effective ways to exploit locality, which is a must to achieve good performance in current computers because of the significant difference in speed between processor and memory. Furthermore, tiles are also useful to express data distribution in parallel computations. However, despite the importance of tiles, most languages do not support them directly. This gives place to bloated programs populated with numerous subscript expressions which make the code difficult to read and coding mistakes more likely.

This paper discusses Hierarchically Tiled Arrays (HTAs), a data type which facilitates the easy manipulation of tiles in object-oriented languages with emphasis on two new features, dynamic partitioning and overlapped tiling. These features facilitate the expression of locality and communication while maintaining the same performance of algorithms written using conventional languages.

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, tiling, locality

1. Introduction

Tiling [2, 20, 23, 33] is ubiquitous in scientific and engineering programs. It is one of the most important strategies for distribution of data and computations in parallel programs [26] and for locality enhancement in parallel and sequential programs [16, 32]. Tiles are of great importance for the implementation of iterative and recursive, serial and parallel algorithms. Tiling is typically planned by the programmer during program design. Sometimes the initial code incorporates tiling and in other cases tiling is applied during program tuning. Some compilers incorporate automatic tiling transformations, but these techniques apply to a limited number of cases and they are not always effective in the tiling of complex algorithms. Language extensions to express tiling are only available in a few parallel languages which include constructs to specify that data should be distributed in blocks across a mesh of processors (HPF [19], UPC [10]). However, even languages, such as Co-Array Fortran [24], that allow explicit references to the blocks in which an array is divided have limited capabilities for the definition and manipulation of tiles. This lack of support for tiling forces the development of programs containing unnecessarily complex code sequences, what decreases readability and increases the likelihood of program defects.

The Hierarchically Tiled Array (HTA) [8] data type is the result of our quest for an easy, thorough and consistent way to embed tiles and their manipulation in sequential and parallel programs. With this data type it is possible to describe in a natural manner arrays partitioned into tiles. In HTAs, tiles can be conventional arrays or lower level HTAs. In [8] we showed how to use HTAs to extend conventional sequential languages to represent parallel computations. In this paper, we present two new class constructs that further help to efficiently express algorithms using HTAs. The first construct, *dynamic partitioning*, allows the modification at run-time of the tiling structure of an HTA in a very flexible way. HTAs extended with this mechanism seem to be the most powerful and easiest way to express an important class of algorithms. The second construct, *overlapped tiling*, hides all the details of the management of shadow regions, which naturally arise when tiling is applied to stencil computations. Our experimental data show that both mechanisms greatly improve some reasonable metrics of programmer productivity, while their impact on performance is negligible.

The rest of the paper is organized as follows. A motivating example will be presented in Section 2. Then, a brief introduction to the HTA data type is presented in Section 3. The two HTA extensions proposed in this paper, dynamic partitioning and overlapped tiling are discussed in Sections 4 and 5, respectively. Section 6 contains our evaluation both in terms of performance and productivity improvement. Finally, a brief discussion on related work in Section 7 precedes our conclusions in Section 8.

2. A motivating example

Tiles are a natural mechanism to express computations with a high degree of locality. In this Section, we use LU decomposition to illustrate programming with tiles [5, 16, 29]. Tiling is of great importance for LU. Our experimental results in Section 6 show that a blocked version of the algorithm with partial pivoting is 20.2 times faster than an unblocked version for a matrix of size 2160×2160 on a 3.0 GHz Pentium 4 machine.

As mentioned in the introduction, a non-trivial effort is required from the programmer in the writing of array subscripts using conventional languages. A good example of the use of tiling for LU is Gustavson's algorithm [16], which stores matrices as a collection of submatrices, each stored in consecutive memory locations, as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

Copyright © 2008 ACM 978-1-59593-960-9/08/0002...$5.00

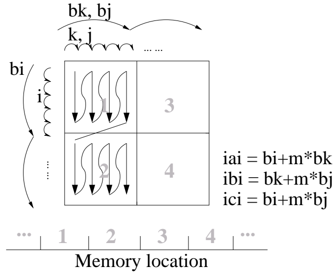


Figure 1. An illustration of NDS in [16]

```
1     subroutine dgefb(M, N, A, b, ipiv, info)
2     ...
3     bk = 0
4     do k=0, N-1, b
5       bj = bk
6       ...
7       do j = k+b, N-1, b
8         bj = bj + 1
9         ibi = bk+m*bj
10         ...
11        bi = bk
12        do i= k+b, M-1, b
13  *
14  *     Update the trailing matrix
15  *
16          bi = bi+1
17          ms = b
18          if(bi .eq. m-1) ms=m2
19          iai = bi+m*bk
20          ici = bi+m*bj
21          call nds_dgemm(ms, ks, b, A(0,0,iai), b,
22     &                 A(0,0,ibi), b, A(0,0,ici), b)
23        enddo
24      enddo
25      ...
26    enddo
27    return
28    end
```

Figure 2. A snapshot of LU factorization extracted from [16]

shown in Figure 1. Gustavson calls this representation New Data Structure (NDS). NDS is a three dimensional FORTRAN array, where the first two dimensions are coordinates within a submatrix or tile and the third dimension is a linearized tile index.

Figure 2 shows a segment of Gustavson's code where NDS is used. In this code, only the loops that control the iteration over tiles along the `i,j, and k` dimensions (lines 4, 7 and 12) and the function call `nds_dgemm` (line 21) are part of the LU algorithm. The other statements and variables are needed to manage the tiles by computing linearized indices `iai, ibi, ici` (lines 9, 19 and 20) as a function of the tile indices for each dimension `bi, bj, bk` (lines 3, 5, 8, 11 and 16), and the tile size (lines 17 and 18). A similar situation arises in the part of the code not shown in Figure 2. In fact, out of 51 lines, 12 statements calculate tile indexes and 11 tile sizes. In other words, 45% of the lines are needed for the bookkeeping of tiles due to the lack of tile support. In the case of LAPACK [3], which implements the same algorithm but with a column major layout, indexing consumes 29 operations that are spread on 11 of the 37 lines (30 lines if lines for checking errors or boundary conditions are excluded) of the routine. A discussion will be presented in Section 6.

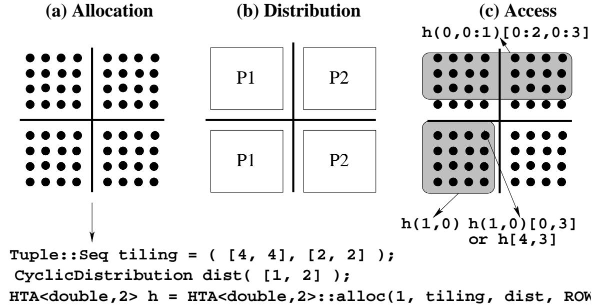


Figure 3. HTA allocation, distribution and access.

3. Hierarchically tiled arrays

Hierarchically Tiled Arrays (HTA) [8] were designed to facilitate programming for parallelism and locality. Classes implementing HTAs have been developed for MATLAB [8] and C++ [9].

Hierarchically Tiled Arrays (HTAs) are arrays that may be partitioned into tiles. These tiles can be conventional arrays or lower level HTAs. Tiles can be distributed across processors in a distributed-memory machine or be stored in a single machine according to a user specified layout.

3.1 Construction of HTAs

Figure 3-(a) shows code for the creation of an HTA with 2×2 tiles of 4×4 elements each. The second parameter of the HTA creation method `alloc` specifies the shape of the HTA. When there are m levels of tiling, the second parameter (`tiling` in the example) must contain $m+1$ tuples, the first is the shape of the innermost (or leaf) tile and the rest are the number of tiles in each level in innermost to outermost order. If the data are distributed across processors, the distribution is specified with the third parameter of the constructor (`dist` in the example), where the mesh of processors and the type of distribution (block, cyclic, block cyclic or user-defined distribution) are specified. Data distribution is illustrated in the second statement of Figure 3 and in in Figure 3-(b). The number of levels of tiling is specified by the first parameter of the constructor (1 in our example), and the data layout by the fourth parameter (the predefined constant `ROW` in the example). The data type and the number of dimensions of the HTA are template parameters of the HTA class.

3.2 HTA data layout and memory mapping

We call the innermost tiles of the hierarchy the HTA leaf tiles. Data in the HTA leaf tiles may have row major (`ROW`), column major (`COLUMN`) or `TILE` layout. The `TILE` layout stores the elements of a tile in continuous memory locations. The elements of each tile are stored in row major order. This is similar to the layout of the NDS data structure described in Section 2 (NDS is column-major whereas our TILE layout is row-major). The layout across tiles is always row-major.

The memory mapping of a distributed HTA is determined by (i) how the tiles of the HTA are allocated in a distributed system, and (ii) the memory layout of the tiles. All this information is determined when the HTA is created: the distribution parameter indicates the home location of the top-level tiles of an HTA, while the last parameter specifies the memory layout.

3.3 Accessing HTAs

HTA indices are 0-based in our C++ implementation. To simplify the notation in this paper, we use the triplet notation `low:step:high` typical of vector languages to represent a range, where `step` can

be omitted if it is 1. Also, a single colon can be used in any index to refer to all the possible values for that index. In our C++ implementation, triplets are represented as `triplet (low,step,high)` or `triplet (low,high)`.

We overload the operator `()` to access tiles and the operator `[]` to index scalar elements in an HTA disregarding the tiling structure. Figure 3-(c) shows examples of how to access HTA components. The expression `h(1,0)` refers to the lower left tile. The scalar element in the fifth row and fourth column can be referenced as `h[4,3]` just as if `h` were an unpartitioned array. This element can also be accessed by selecting the tile that contains it and its relative position within this tile: `h(1,0)[0,3]`. In any kind of indexing, a range of components may be chosen in each dimension using triplets notation. For example, `h(0,0:1)[0:2,0:3]` selects the first four rows in tiles (0,0) and (0,1). When an HTA contains multiple levels of tiling, it can be indexed using several tuples, one for each level.

3.4 Binary operations, assignments and methods

We generalize the notion of conformability of Fortran 90. When two HTAs are used in an expression, they must be conformable, i.e., they must have the same topology and the corresponding tiles in the topology must have sizes that allow to operate them. The operation is executed tile by tile, and the output HTA has the same topology as the operands.

Also, an HTA is always conformable to a scalar. The scalar operates with each scalar component of the HTA. An HTA is also conformable to an untiled array if each HTA leaf tile is conformable with the array. The untiled array will be operated with each HTA leaf tile.

Assignments to HTAs 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) if both are conformable. The number of levels of tiling assumed for operations and assignments could be controlled by the number of subscript tuples used in the HTAs. Thus, an HTA with m levels of tiling would be assumed to have only $n < m$ levels of tiling if the HTA name is followed by only n subscript tuples within `()`s and one subscript tuple enclosed within `[]`s. In particular, an HTA would be seen as an untiled array, if the name of the HTA is followed immediately by a `[]` tuple as illustrated in Figure 3-(c).

References to local HTAs do not involve communication. However, in distributed HTAs assignments between tiles located in different processors involve communication.

We have implemented frequently-used array functions such as `circshift`, `transpose`, `permute`, or `repmat`, and have overloaded the standard arithmetic operators (+, -, *, /) so that when applied to HTAs they operate at the tile level. This helps to hide the loops and scalar accesses within a tile and to express the algorithm at the tile level. When the HTA is a distributed HTA, array functions like the ones just mentioned applied at the level where the HTA is distributed result in cross-processor communication.

3.5 Implementation

The C++ implementation of the HTA class is a library with ∼18000 lines of code, excluding comments and empty lines. It only contains header files, as most classes in the library are C++ templates to facilitate inlining. Our implementation uses the optimization techniques that C++ templates enable and that are of particular interest for scientific computing, such as expression templates, traits and static polymorphism [30]. In this way we avoid the usually costly virtual function calls despite the fact that our implementation relies heavily on operator overloading and polymorphism.

```
1  typedef HTA<double,2> HTA2;
2  A = HTA2::alloc(2,([ts, ts], [nt1, nt1], [nt2, nt2]), TILE);
3  ... ...
4  void  mult(HTA2 A, HTA2 B, HTA2 C)
5  {
6    if(A.level()==0){
7      ATLAS_mini_MMM(A, B, C);
8    } else {
9      for (int i = 0; i< A.size(0); i++)
10       for (int k = 0; k < B.size(0); k++)
11         for (int j = 0; j< C.size(0); j++)
12           mult(A(i,k), B(k,j),C(i,j));
13   }
14 }
```

Figure 4. Recursive blocked matrix multiplication using HTAs

The library has been designed in a modular way in order to allow flexibility in the choice of the underlying communication and threading mechanisms. There are a series of classes implementing abstract functionalities on the data type, some of which must be specialized to enable execution on an specific backend. Our stable implementation currently supports sequential execution as well as parallel execution on top of MPI [15] and Intel Threading Building Blocks [27]. An experimental implementation based on the UPC run-time library targeted by the IBM XL UPC compiler [4] has also been developed. The backend is selected at compile time by means of an user-defined constant. HTA codes do not need to be changed when moving to different backend.

3.6 A natural representation for blocked algorithms

We now discuss how the HTA data type helps to represent the tiling and data layout in different algorithms. We show examples of HTA programs for matrix matrix multiplication and LU factorization.

3.6.1 Matrix matrix multiplication

Figure 4 shows an HTA implementation of a recursive blocked matrix matrix multiplication algorithm. HTAs `A`, `B`, and `C` are declared and initialized outside the `mult` function. The creation of `A` is shown in line 2; `B` and `C` are created in a similar way. The first parameter in `alloc` specifies two levels of tiling for `A`. The second parameter, an array of tuples, provides the tile size for each level. The last parameter determines the data layout to be `TILE`.

In the recursive function `mult`, if the input HTA `A` is just an array (i.e. is a leaf of an HTA), it calls the matrix matrix multiplication code (line 7), which is optimized for small matrices that fit into the cache. Otherwise, the corresponding tiles of `A` and `B`, which are also HTAs, will be multiplied by calling `mult` recursively. Notice that for efficiency the HTA objects passed to the functions are references to the internal structure of the HTA. Thus, changes using the HTA arguments of the function will modify the original HTA.

The configuration of an HTA object is easy to manipulate. Modifying the levels of tiling, the tile size, and the data layout can be simply done by changing the values used in `alloc`. No changes to the code are required, i.e., adding an extra level of tiling does not require adding additional loops. Thus, programmers can tune an application by simply trying different values for the parameters of `alloc`.

3.6.2 LU factorization

LU decomposition factorizes a matrix, say A, into the product of a lower (L) and upper(U) triangular matrix such that $A = LU$. In this Section, we first present the LU algorithm in LAPACK [3] and described by Gustavson [16]. We denote this algorithm *iterative*

Step 1. Factor
$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}$

Step 2. Permute
$\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \leftarrow P_1 \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}.$

Step 3. Permute
$\begin{bmatrix} L_{10} \\ L_{20} \end{bmatrix} \leftarrow P_1 \begin{bmatrix} L_{10} \\ L_{20} \end{bmatrix}.$

Step 4. Solve
$L_{11}U_{12} = A_{12}$

Step 5. Update
$A_{22} \leftarrow A_{22} - L_{21}U_{12}$

```
1 void LU(HTA<double, 2> A,HTA<int,1> p)
2 {
3  int m = A.size(0)-1;
4  int n = A.size(1)-1;
5
6  for( int k=0; k<=n; k++){
7    //step 1 Factor
8    hta_getf2(A(k:m, k), p(k));
9
10   if(k<n){
11     //step 2 Permute
12     hta_laswp(A(k:m, k+1:n), p(k));
13
14     //step 3 Permute backward
15     hta_laswp(A(k:m, 0:k-1), p(k));
16
17     //step 4 Solve triangular system
18     hta_trsm(Right, Upper, NoTrans,
19       Unit, 1.0, A(k), A(k, k+1:n));
20   }
21   if(k>1)
22     //step 5 Update trailing matrix
23     hta_gemm(NoTrans, NoTrans, -1.0,
24             A(k+1:m,k), A(k, k+1:n),
25               1.0,  A(k+1:m, k+1:n));
26  }
27}
```

Figure 5. Iterative static LU factorization algorithm (left) and its HTA implementation

static LU algorithm since it pre-partitions the array before factorization.

The *iterative static* LU algorithm factors an $M \times N$ matrix A into an $M \times M$ permutation matrix P, an $M \times N$ unit lower triangular matrix L, and an $N \times N$ upper triangular matrix U, such that $PA = LU$, where $M \geq N$. A is decomposed into a $m \times n$ array of tiles T_{ij} of size $b \times b$, where $0 \leq i \leq m-1, 0 \leq j \leq n-1$. In the kth iteration, we group the tiles into 9 collections:

$$PA = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}.$$

where A_{11} is T_{kk}, A_{00} contains all the tiles in the upper left corner. A_{22} contains all the tiles in the bottom right corner. Matrices P, L, U are partitioned and grouped in a similar way. In the kth iteration, the algorithm performs the five steps shown on the left of Figure 5.

The HTA implementation is shown on the right of Figure 5. The input and output parameters `A` and `p` are one level HTAs. They are statically partitioned and passed to function `LU`. Each step inside of the iteration is completed by one function call. The notation `hta_`*func* is the HTA version of *func*, a BLAS or LAPACK library function. Notice that there is no need to explicitly pass other information such as the data type of the arrays, their sizes, the length of the leading dimensions, etc. in the invocations because the HTAs already carry it. In this way, the programmer can focus more on the algorithm itself rather than the parameter details. If necessary, programmers can quickly write an HTA wrapper function since the information related to tile and data layout can be easily derived from the HTA data type.

Compared to Figure 2, Figure 5 demonstrates the clean syntax provided by HTAs as follows. First, the use of multi-dimensional tile indexing avoids unnecessary variables and extra indexing calculations. Second, access to a range of tiles can be expressed in HTA programs using the triplet notation. For example, `A(k:m,k)` in line 8 is a tile panel which consists of tile `A(k,k)` and the tiles below it. Such vectorized code eliminates loops, and makes the control flow of the program simpler. Finally, as discussed above, subroutine interface is greatly simplified.

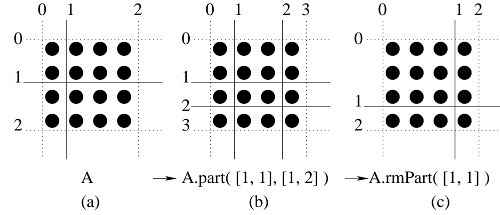


Figure 6. Illustration of dynamic partitioning in HTAs

4. Dynamic partitioning

All the examples in Section 3 assume *static partitioning*: the shape of the tiles is fixed at HTA creation time and it does not change throughout the program. However, many algorithms such as cache oblivious algorithms [14] or many of the algorithms produced by FLAME [5] require dynamic changes of the tile layout. FLAME is a system developed at the University of Texas which introduces a new notation for expressing linear algebra algorithms and a methodology for their systematic derivation. The need for dynamic partitioning motivates the extension of the HTA data type with new language operations to change the tiling structure of the HTAs on the fly.

4.1 Syntax and Semantics

We first describe the partition numbering scheme we use in the syntax of dynamic partitioning. In an HTA, the lines that separate the tiles are called *partition lines*. There are implicit partition lines before the first and after the last element in each dimension. Other partition lines can be specified by the user. We number the partition lines starting with the one before the first element, which is numbered 0. For example, in Figure 6-(a), the partition lines are numbered from 0 to 2 (dotted lines are implicit partitions and solid lines are user-specified partitions).

A set of partition lines, containing no more than one line for each dimension of an HTA, is called a *partition*. A partition is represented by a tuple that for each dimension contains the partition number or the special symbol `NONE` when there is no partition line for a given dimension. Since in most cases changing the tiling structure requires adding only one partition, we provide operations that add (`part`) or remove (`rmPart`) one partition. Their syntax is as follows:

```
void part(Tuple sourcePartition, Tuple offset);
void rmPart (Tuple partition);
```

Let $\texttt{sourcePartition}_i$ be the i-th element in `sourcePartition`, `part` adds a new partition line along the i-th dimension of the HTA at location Pos($\texttt{sourcePartition}_i, i$) + $\texttt{offset}_i$, where Pos(P,d) is the location of the partition line P in the dimension d. No partition line is added if $\texttt{sourcePartition}_i$ is `NONE`. Notice that $\texttt{offset}_i$ can be positive or negative.

Figure 6-(b) shows an example of the use of `part` and `rmPart`. A request is made to create a new partition with offset `(1,2)` relative to partition `(1,1)`. Notice that since partition lines are numbered consecutively the new partition will be identified as `(2,2)` and the implicit partition lines at the end of the array will be numbered 3 in each dimension.

To remove a partition we use the method `rmPart` with the tuple that identifies the partition to be removed. Only explicit partition lines can be removed. Calling `rmPart` with no arguments removes all the explicit partitions. Figure 6-(c) illustrates the removal of partition `(1,1)` from the HTA resulting from Figure 6-(b). Notice that after eliminating a partition, the partition lines are renumbered.

Algorithm: $[A, p] := LU_{PIV_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$

where A_{TL} and p_T are empty

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} is $b \times b$, p_1 is $b \times b$

$$[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1] := LU_{PIV}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := L_{11}^{-1} A_{12}$$

$$A_{22} := A_{22} - A_{21} A_{12}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figure 7. The iterative dynamic LU factorization derived from FLAME

This partition scheme allows programmers to perform dynamic partitioning at the tile level, without the need of knowing the absolute positions in the array of the partition delimiters. Finally, notice that our current implementation changes a single partition at a time. While it is possible to implement a mechanism to re-partition the whole array at once, we have not implemented it because so far we have not found algorithms that could benefit from this.

4.1.1 Empty tiles

When the method `part` is called to add a new partition line with offset 0 with respect to an existing partition line, we could have assumed that either a new partition line will be added or not. Since the offset is zero, adding a new partition line means adding an empty tile between the original line and the new one. Our implementation adds the partition line because with empty tiles the writing of some array algorithms is simplified. We show an example that makes use of empty tiles in Section 4.2.

Operations and assignment of HTAs with empty tiles also follow the rule of conformability. That is, empty tiles cannot be operated with or be assigned to non-empty tiles, since the shapes are not equal. The methods that take empty tiles as input will immediately exit since they do not contain data.

4.2 Examples

In this section, we study the use of dynamic partitioning to represent algorithms derived from FLAME [5] and cache oblivious algorithms [14].

4.2.1 Representation of FLAME-generated algorithms

The FLAME project [5] uses a systematic approach to derive families of high-performance algorithms for dense linear algebra algorithms. We illustrate FLAME's notation in Figure 7, using blocked LU factorization with partial pivoting. The algorithm is divided into three parts by solid horizontal lines. In the first part, arrays A and p are partitioned so that empty tiles appear at the top and left. At the

```
1   void lu(HTA<double,2> A, HTA<int,1> p,int b)
2   {
3     A.part( [0, 0], [0, 0] );
4     p.part( [0],  [0] );
5
6     while( A(0,0).lsize(1) < A.lsize(1) ){
7       A.part( [1, 1], [b,b] );
8       p.part( [1], [b] );
9       hta_getf2(A(1:2,1), p(1));
10      hta_laswp(A(1:2,0), p(1));
11      hta_laswp(A(1:2,2), p(1));
12      hta_trsm(Right, Upper, NoTrans,
13               Unit, 1.0, A(1,1),A(1,2));
14      hta_gemm(NoTrans, NoTrans, -1.0,
15               A(2,1), A(1,2), 1.0, A(2,2));
16      A.rmPart( [1, 1] );
17      p.rmPart( [1] );
18    }
19  }
```

Figure 8. HTA implementation of the iterative dynamic LU algorithm in Figure 7

beginning of the `while` loop body, A and p are updated by adding new partitions at the bottom for p and at the right and bottom for A. The T, B, L, R in subscripts stand for Top, Bottom, Left, and Right, respectively. The update for the newly created tiles is performed in the second part of the algorithm. The third part removes the old partitions for the current iteration and the remaining partition becomes the old one in the next iteration (it removes the partitions represented with thin lines). Finally, the desired result has been computed when A_{TL} encompasses all of A making the loop condition false. Here $n(A)$ returns the number of columns of A.

The algorithm displays the stages found in all FLAME-derived algorithms [5]: *partition*, *repartition* and *continue with*. These stages, which appear in the first and the third part of the algorithm determine how the algorithm sweeps through the matrices.

The actions in the first and third part can be straightforwardly represented using dynamic partitioning. Using `part` for both *partition* and *repartition* and `rmPart` for *continue with*, we can have a one to one mapping from the actions described in the FLAME algorithms to the HTA methods, as shown in the code in Figure 8. Here `A` and `p` are HTAs with a single tile when function `lu` is called, and `lsize` is an HTA method that returns a tuple with the number of elements in each dimension of the corresponding HTA. As it can be seen, the HTA program does not use as many different variables as in the algorithm description from FLAME, since tiles can be easily accessed in HTA programs through tile indexing, by either referencing a single tile or a range of tiles.

Section 6 compares, both in terms of performance and readability, the implementation of algorithms using HTAs and the API of the FLAME project [6], which closely mimics the notation in Figure 7.

4.2.2 Representation of cache oblivious algorithms

Cache oblivious algorithms [14] are usually recursive algorithms following a divide-and-conquer strategy. Next, we present a recursive LU decomposition algorithm proposed by Toledo [29], as an example of cache oblivious algorithm. Figure 9 shows the algorithm, with matrices A, p and L being the same as those in FLAME.

In this algorithm, if A has more than one column, it is partitioned in half along both dimensions and the factorization is performed recursively on the left half until a single column is obtained. The factorization of a single column takes place in function LU_{INNER}. When the algorithm returns from the recursive call at (*), the factorization for the left half of the portion of A has already been completed, and it proceeds to apply the appropriate permutation and updates to the right half of the matrix. Then another

Algorithm: $[A, p] := LU_{REC}(A)$
if $n(A) = 1$
$[A, p] = LU_{INNER}(A)$
else
Partition $A \rightarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right)$, $p \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \end{array}\right)$
where A_{00} is $n/2 \times n/2$, p_0 has $n/2$ elements

$$\left[\left(\frac{A_{00}}{A_{10}}\right), p_0\right] := LU_{REC}\left(\frac{A_{00}}{A_{10}}\right) \quad (*)$$
$$\left(\frac{A_{01}}{A_{11}}\right) := P(p_0)\left(\frac{A_{01}}{A_{11}}\right)$$
$$A_{01} := L_{00}^{-1} A_{01}$$
$$A_{11} := A_{11} - A_{10} A_{01}$$
$$[A_{11}, p_1] := LU_{REC}(A_{11}) \quad (**)$$
$$A_{10} := P(p_1)(A_{10})$$

endif

Figure 9. Toledo's recursive LU factorization [29]

```
1  void lu(HTA<double,2> A, HTA<int,1> p)
2  {
3    int m = A.lsize(0);
4    if( m <= 1) {
5       lu_inner (A, p);
6    } else {
7       int b = m/2;
8      A.part( [NONE, 0], [NONE, b] );
9      p.part( [0], [b] );
10
11     lu(A(0,0), p(0));
12     hta_laswp(A(0,1), p(0));
13
14     A.part( [0,NONE], [b,NONE] );
15
16     hta_trsm(Right, Upper,NoTrans,
17        Unit, 1.0, A(0,0), A(0,1));
18     hta_gemm(NoTrans, NoTrans, -1.0,
19       A(1,0), A(0,1), 1.0, A(1,1));
20
21     lu(A(1,1), p(1));
22     hta_laswp(A(1,0), p(1));
23     p(1) += b;
24
25     A.rmPart();
26     p.rmPart();
27   }
28 }
```

Figure 10. HTA implementation of Toledo's recursive LU factorization [29]

recursive invocation at (**) is needed to factor A_{11}. Once this is achieved, the permutation obtained P_1 must be applied to A_{10}, after which the results are ready.

One of the challenges to implement the algorithm is to determine how to partition the A matrix. If we were to partition A both vertically and horizontally in half at the beginning, the recursive function call in (*) will carry the horizontal partition line into the next level of the recursion. Once we reach a single column, the number of horizontal partition lines added would be equal to the number of recursions performed in (*). Adding horizontal partition lines at the beginning makes it difficult to index the column panels. However, using dynamic partitioning we can add the horizontal partition lines later, when they are needed.

Figure 10 shows our implementation of Toledo's recursive LU factorization algorithm. It first partitions A vertically in line 8, and the left half of A is recursively factorized in line 11. The horizontal partition is added in line 14. At the end of the function, all the partitions created in the current recursion are removed to avoid interfering with outer invocations of the recursive routine and to save storage space for tiling structures.

```
1 struct PMerge {
2   void operator() (HTA1 out, HTA1 in1, HTA1 in2) {
3     int in1_size = in1.lsize(0);
4     if (in1_size > GRAINSIZE) { /* parallel merge */
5       int midpos_in1 = in1_size / 2 ;
6       int cutpoint_in2 = in2.lowerBoundPos(in1[midpos_in1]);
7       in1.part( [0], [midpos_in1] );
8       in2.part( [0], [cutpoint_in2] );
9       out.part( [0], [midpos_in1  + cutpoint_in2] );
10      map(PMerge(), out, in1, in2);
11      in1.rmPart();
12      in2.rmPart();
13      out.rmPart();
14    } else {
15       /* sequential merge */
16    }
17  }
18 };
```

Figure 11. Parallel merge using HTA dynamic partitioning

Another example of how dynamic partitioning helps express cache oblivious algorithms and at the same time enable parallelism is the parallel merge shown in Figure 11. The algorithm is implemented in the `operator()` member function of `struct PMerge`. Its input are the HTAs `in1` and `in2`, each containing one of the sorted sequences to be merged. HTA `out` contains the merged sequence at the end of the algorithm. The three HTAs are unidimensional and initially have a single tile.

When the number of elements in `in1` is larger than `GRAINSIZE`, all three HTAs are partitioned into two tiles. Corresponding tiles are then merged in parallel with each other. Otherwise, the operation is performed sequentially. In order to perform the partitioning, the middle element in `in1` is chosen, and the first position in `in2` with a value greater or equal than this value is found using method `lowerBoundPos`. This gives us the location where `in2` must be partitioned. The output HTA is then partitioned into two tiles sized to accommodate the result of the merge of the corresponding tiles in `in1` and `in2`. Function `map` applies a functor in parallel to the corresponding tiles of a set of HTAs. In our case, it generates two parallel invocations of the `PMerge` functor: one on `out(0)`, `in1(0)` and `in2(0)`; and another one on `out(1)`, `in1(1)` and `in2(1)`. Both can run straightforwardly on different threads in a shared memory multiprocessor/multicore, while communication of tiles would be needed in a distributed memory environment. `map` returns only when both parallel merges have finished. Then the partitions are removed from the HTAs.

Figure 12 shows the same algorithm implemented using the Intel Threading Building Blocks (TBB) [27], a library that helps express parallel computations in shared memory systems. The parallelism here is given by the `parallel_for` function in line 29, which receives a `PMergeRange` representing the problem range on which to operate and a functor `PMergeBody` that performs the sequential merge on a range of data. The information required to represent a range of the problem are the beginning and ending iterators (type `It` in the code) that mark the limits of the two input arrays (`begin1`, `end1`, `begin2`, `end2`) and an iterator `out` pointing to the beginning of the output array. All of them are stored inside a `PMergeRange`. TBB ranges must provide methods `empty` and `is_divisible` to inform where they are empty and whether they can be decomposed further, respectively. TBB's `parallel_for` obtains parallelism by decomposing the range provided in smaller ranges and processing them in parallel using the task functor, `PMergeBody` in this case. When TBB decides to decompose a `PMergeRange` to obtain more parallelism, the constructor in line 9 in the code is invoked. This constructor receives a `PMergeRange r` that is to be partitioned in two. After the partition, `r` will store the first half of the original range it contained,

```
1  struct PMergeRange {
2    It begin1, end1; // [begin1,end1) 1st sequence to merge
3    It begin2, end2; // [begin2,end2) 2nd sequence to merge
4    It out;           // where to put merged sequence
5    bool empty() const
6       { return (end1-begin1)+(end2-begin2)==0; }
7    bool is_divisible() const
8       { return (end1 - begin1) > GRAINSIZE; }
9    PMergeRange( PMergeRange& r, split )
10    : end1(r.end1), end2(r.end2)
11   {
12     begin1 = r.begin1 + (r.end1 - r.begin1)/2;
13     begin2 = lower_bound(r.begin2, r.end2, *begin1);
14     out = r.out + (begin1 - r.begin1) + (begin2 - r.begin2);
15     r.end1 = begin1;
16     r.end2 = begin2;
17   }
18   PMergeRange(It b1, It e1, It b2, It e2, It out1)
19    : begin1(b1), end1(e1), begin2(b2),end2(e2),out(out1) {}
20 };
21
22 struct PMergeBody {
23   void operator() (PMergeRange& r) const {
24     /* sequential merge */
25   }
26 };
27
28 void PMerge(It begin1,It end1,It begin2,It end2,It out) {
29  parallel_for(PMergeRange(begin1,end1,begin2,end2,out),
30               PMergeBody() );
31 }
```

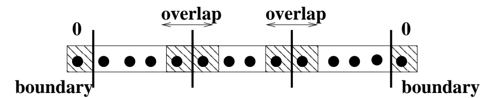
Figure 12. Parallel merge using Intel Threading Building Blocks

while the newly constructed PMergeRange will store the second half. This way, lines 12 and 13 perform the same function as lines 5 and 6 in Figure 11, that is, finding the point in which the first and second array must be partitioned respectively. As we see, TBB requires more lines of code, variables, and data types than the HTA to express the same problem.

5. Overlapped tiling

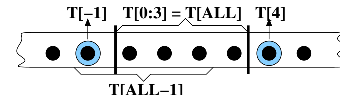
Iterative finite difference techniques perform neighbor based computations called *stencil* computations. In a stencil operation, each point in a multi-dimensional array is computed based on the weighted value of its neighbors. Stencil codes benefit from tiling, because they increase locality and determine data distribution when running in parallel. Stencil codes compute new values based on neighboring values. As a result, when computing the values inside a tile, some elements of the neighboring tiles must be accessed. When running stencil codes in a distributed memory machine, accessing the neighboring tiles requires remote accesses. To solve this problem, programmers create a shadow or ghost region around each tile that contains a copy of the elements of the neighbor tiles needed for the computation. Ghost regions are useful in sequential codes when the data is stored by tiles to simplify index calculations.

Shadow regions distort the implementation of stencil algorithms, because they must be updated to reflect the changes performed on the values they replicate. This results in complex codes that are usually much longer than the original non-tiled stencil computation. For example, the NAS MG benchmark [1], that performs a multi-grid computation on a discrete 3D grid with periodic boundaries, contains a routine comm3 that updates the shadow regions of most of the stencils of the program. In the MPI [15] implementation of MG, comm3 and the routines it invokes are 327 lines long. In contrast the computational part of the stencils of MG is much shorter. The three stencils that use comm3 require 96 lines (114 with debugging code): 23 in resid, 24 in psinv and 49 in rprj3. Implementations of shadow regions in languages that provide a global view of the data are not much better: the Co-array




```
Tuple::seq tiling = ( [4], [3] );
Overlap<1>   ol( [1], [1], zero );
A = HTA<double,1>::alloc(1, tiling, array, ROW, ol);
```

(a)



(b)

Figure 13. Example of overlapped tiling. (a) Construction of an HTA with tiles that overlap in both directions and boundary regions. (b) Indexing for overlapped HTA

Fortran [24] version of comm3 and the routines it invokes needs 302 lines. Even in the serial code or in the OpenMP [11], where the shadow regions are necessary to store the periodic boundaries, the amount of extra code is significant. In the serial version of MG, comm3 is 29 lines long, while the OpenMP version of comm3 is 26 lines long. Overlapped tiling makes comm3 unnecessary.

With overlapped tiling, the programmers can specify at HTA creation time that they want the tiles of the HTA to overlap to a given extent, and that they want the shadow regions to be kept updated.

5.1 Syntax and semantics

That an HTA is to have an overlapped tiling organization is specified with a fifth optional parameter in the alloc HTA constructor. This parameter must be an object of the class Overlap. This object conveys the information about overlapped tiling requested by the user which can be built using the constructor

```
Overlap<DIM> ol = Overlap<DIM>( Tuple negativeDir,
                                Tuple positiveDir,
                                boundaryMode mode,
                                bool autoUpdate = true);
```

where DIM is the number of dimensions of the HTA to be created. The negativeDir specifies the amount of overlap for each tile in the *negative direction* (decreasing index value) in each dimension. The positiveDir specifies the amount of overlap for each tile in the *positive direction* (increasing index values). Thus these parameters define the size of the overlap and the size of the boundary region that needs to be built around the array. The third argument specifies the nature of this boundary region and its value can be either zero or periodic. In the first case, the boundary region will be filled with constants, while in the second case the boundary will be periodic, i.e, it will replicate the values of the array in the opposite side. Finally, the fourth (optional) argument specifies whether the library must automatically update the shadow regions. Its default value is true.

Figure 13-(a) illustrates the creation of a 1-dimensional HTA with three tiles and four elements per tile. There is an overlap of one element between the tiles in each direction. The boundaries have value zero. The overlapping extends the length of each tile to enable access to the boundary elements from neighboring tiles. The range of index values of each tile is from -1 to 4, both included, as shown in Figure 13-(b). The symbol ALL may be used to refer to the elements of the tile (range 0:3 in our example). Constants

```
1  int ntiles = N / d;
2  Tuple::seq tiling =( [d+2, d+2], [ntiles, ntiles] );
3  v = HTA<double, 2>::alloc(1, tiling, dist, ROW);
4  u = HTA<double, 2>::alloc(1, tiling, dist, ROW);
5  ...
6  while( dif < epsilon) {
7    v(1:ntiles - 1, :)[0, :]     = v(0: ntiles - 2, :)[d, :];
8    v(0:ntiles - 2, :)[d + 1, :] = v(1:ntiles - 1, :)[1, :];
9    v(:, 1:ntiles - 1)[:, 0]     = v(:, 0: ntiles - 2)[:, d];
10   v(:, 0:ntiles - 2)[:, d + 1] = v(:, 1:ntiles - 1)[:, 1];
11
12   u(:,:)[1:d, 1:d] = k * (v(:,:)[1:d, 0:d - 1] +
13                           v(:,:)[0:d - 1, 1:d] +
14                           v(:,:)[1:d, 2:d + 1] +
15                           v(:,:)[2:d + 1, 1:d] );
16    dif = maxdif(u(:,:)[1:d][1:d], v(:,:)[1:d][1:d]);
17    v = u;
18 }
```

Figure 14. Stencil code with explicit shadow region exchange

can be added or subtracted from ALL to shift the indexed area as illustrated in Figure 13-(b).

Notice that conformability rules only apply to the tile itself without including the shadow regions. Also assigning to a tile can only change the elements of the tile and not those of the neighboring tiles.

5.1.1 Consistency

As explained before, programmers can request the system to automatically update the shadow regions or they can perform the update manually. When automatic update is chosen, the HTA library follows an *update on read* policy, which consists in updating the shadow regions when the processor where they reside reads them. We chose this approach over *update on write* because *update on read* results in fewer copies/communications, since data is copied only if it is actually going to be used.

When the user chooses manual update, our implementation provides some methods: `update` to perform a synchronized update, and `asyncUpdate` and `sync` to perform asynchronous updates, which allow to overlap communication and computation. A situation in which programmers may choose manual updates is to avoid delayed updates and enable communication and computation overlap. Also it is desirable when updates are not required, which happens with some temporaries.

5.2 Example

Let us examine a Jacobi relaxation code on an $N \times N$ matrix using tiles of size $d \times d$. An implementation of this algorithm using HTAs without overlapped tiling is shown in Figure 14. Each block is represented by a tile in the HTAs. The tiles are actually of size $(d+2) \times (d+2)$ in order to accommodate the extra rows and columns that act as shadow regions. Border exchange is executed in the first four statements of the main loop. The actual computations only use local data.

The version with overlapped tiling is shown in Figure 15. It does not need to declare explicitly the shadow regions or update them. With the `ALL` construct the update could have the simple form `u = k * (v(:,:)[ALL - [0, 1] ] + ...)`.

6. Evaluation

We now examine the validity of our approach to write codes using the HTA data type both in terms of performance and readability, focusing on the two new language constructs introduced in this paper. For most benchmarks we observe little performance degradation compared to the hand optimized codes. This is because the additional cost of the functions to create and manipulate tiles can

```
1  int ntiles = N / d;
2  Tuple::seq tiling =( [d, d], [ntiles, ntiles] );
3  Overlap<2> ol( [1, 1], [1, 1], zero);
4  v = HTA<double, 2>::alloc(1, tiling, dist, ROW, ol);
5  u = HTA<double, 2>::alloc(1, tiling, dist, ROW, ol);
6  ...
7  while( dif < epsilon) {
8    u = k * (v(:,:)[0:d - 1, -1:d - 2] +
9             v(:,:)[-1:d - 2, 0:d - 1] +
10            v(:,:)[0:d - 1, 1:d] +
11            v(:,:)[1:d, 0:d - 1] );
12    dif = maxdif(u, v);
13    v = u;
14 }
```

Figure 15. Stencil code using overlapped tiling

be amortized over enough computations for each tile. On the other hand, readability is improved greatly thanks to the simplification of indexing, function interfaces, and concise representation of the algorithms.

6.1 Sequential benchmarks

We run our sequential benchmarks on a 3.0 GHz Intel Pentium 4 with 16KB L1, 1MB L2 and 1GB RAM. We used the g++ compiler version 3.2.3 with -O3 optimization flag both for the HTA programs and the hand optimized C++ codes.

6.1.1 Matrix matrix multiplication

First, we show the performance of the Matrix Matrix Multiplication in Figure 16. The HTA code used for the experiments is shown in Figure 4. We studied the effect of different parameter values at tile creation in line 2 of Figure 4, including: different number of levels in `A`, `B` and `C` (denoted by `Lvl: n` in the legend of Figure 16); different data layouts for HTA `A` and `B` (denoted by `ROW` or `TILE`). HTA `C` always has a `ROW` data layout. The tile size for the HTA leaves was always 36×36, the value that ATLAS [31] found to be optimal for this platform. When 2-level tiling is used, the middle level tiles contain 4×4 leaves each. We compared the performance of the different HTA versions versus ATLAS [31] version 3.2.3 and versus an untiled version consisting of three nested loops. 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. Our results show that the performance of the HTA code using one level of tiling and TILE layout performs almost as well as ATLAS except for smaller matrices (sizes 288×288 and 576×576, where the overhead of recursion becomes more important. Thanks to the HTA notation, it took us very little effort to search for the levels of tiling and memory layout that delivered the best performance.

6.1.2 LU decomposition

We show the performance of the iterative static LU algorithm, iterative dynamic LU algorithm, and the recursive LU algorithm in Figures 17, 18 and 19, respectively. The tile size used was 72×72 for the iterative static and dynamic LU algorithms. For the recursive algorithm, for performance reasons we stopped when A had 4 columns instead of 1 column. The HTA codes for the three algorithms are the ones shown in Figures 5, 8 and 10, respectively.

We examined the performance of the HTA iterative LU algorithm with the data layouts `COL` and `TILE`, and compared them with the LU performance from LAPACK [3]. LAPACK implements the same iterative LU factorization, with column major layout for the matrix A. Figure 17 shows that the version `HTA COL` has a negligible performance difference from the LAPACK version.

The iterative dynamic LU algorithm, was implemented using both HTAs and the FLAME C API [6]. Since FLAME derives a

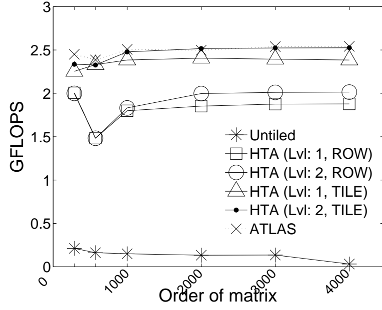


Figure 16. Performance of MMM

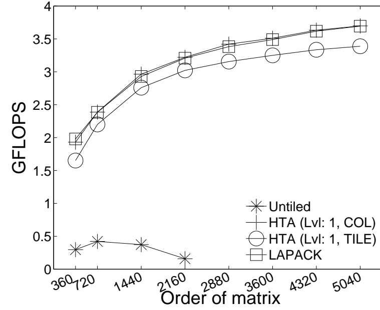


Figure 17. Performance of iterative static LU algorithm

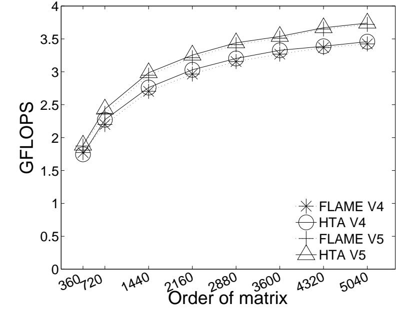


Figure 18. Performance of iterative dynamic LU algorithm

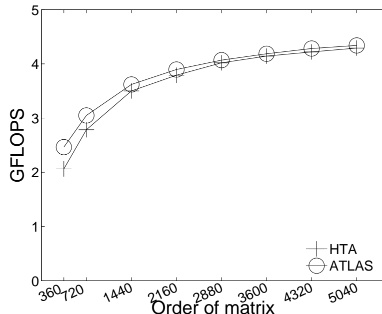


Figure 19. Performance of recursive LU algorithm

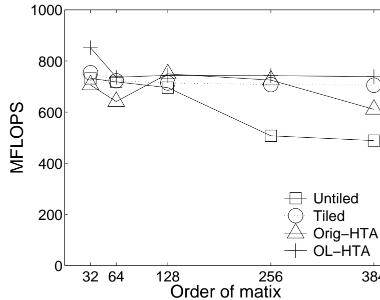


Figure 20. Performance of sequential 3D Jacobi computation

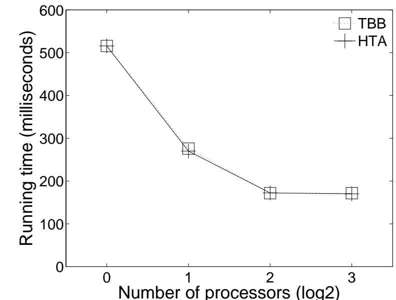


Figure 21. Performance of parallel merge

family of variations for LU factorization, we chose two versions, `v4` and `v5`, to compare. Version `v5` is shown in Figure 7. The performances of HTA and FLAME API are very close, as shown in Figure 18. Further, HTA's performance is within 1.6% difference in average from that of LAPACK.

Figure 19 shows two implementations of the LU recursive cache oblivious algorithms: our HTA version from Figure 10 and the library routine from ATLAS, which follows the same recursive implementation. Again, the average performance difference between the HTA code and the ATLAS routine is just 2.7%.

To summarize, Figures 17, 18 and 19 show little performance degradation using HTA's static or dynamic partitioning.

6.1.3 3D Jacobi

Figure 20 compares a naive C++ implementation of 3D Jacobi (Untiled), a tiled one (Tiled), an HTA program without overlapped tiling (Orig-HTA), and an HTA program with overlapped tiling (OL-HTA). We searched for the best tile sizes for the three tiled versions and present the one that obtained the best performance. Figure 20 shows that the performance of the untiled version drops when the matrix size increases to 256^3 due to poor data locality. The `Tiled` and `OL-HTA` versions do not suffer from performance degradation thanks to tiling. The `Orig-HTA` performs worse than `OL-HTA` due to the overhead of managing the shadow regions to access the elements across the neighbors. Compared with the tiled version, the `OL-HTA` version obtains similar performance and is even slightly better. This is because while the tiled version uses three outer loops to iterate 3D tiles, `OL-HTA` uses a single loop to traverse the tiles.

6.2 Parallel benchmarks

6.2.1 Parallel Merge

Figure 21 compare the performance of parallel merge using TBB [27] and its corresponding HTA version, as shown in the codes Figures 12 and 11, respectively. The two codes run in a system with two Quad core 2.66 GHz Xeon processors in order to evaluate the performance of our dynamic partitioning implementation in parallel executions. Both versions were compiled with g++ compiler version 4.1.2 and O2 optimization level. For this experiment two sequences of 50 million floats each using a grain size of one million elements were merged. As Figure 21 shows, our HTA code (that uses the TBB backend for this experiment) runs as fast as the TBB code.

6.2.2 MG and LU NAS benchmarks

We rewrote the NAS [1] MG and LU benchmarks using HTAs with overlapped tiling in order to evaluate the performance of this operator for parallel computations. The experiments were made in a cluster consisting of 128 nodes each with two 2 GHz G5 processors and 4 GB of RAM. We used one processor per node in our experiments. The primary network connecting the cluster machines is a high-bandwidth, low-latency Myrinet network. The NAS codes were compiled with g77, while the HTA codes were compiled with g++ compiler version 3.3. O3 optimization level was used in both cases. For these experiments the HTA codes used the MPI backend.

The MG benchmark, as well as the parallel Jacobi program, represents examples of loosely synchronous computations, where all the processors alternate between phases of local computation and synchronous global communications [13]. In the NAS MG pro-

```
1   //NX, NY, NZ are the number of tiles in dimensions x,y,z
2   //nx, ny, nz are the tile sizes in dimensions x,y,z
3   if (NX > 0)
4      u(0:NX-1, 0:NY, 0:NZ)[nx, 1:ny-1,1:nz-1] =
5             u(1:NX, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
6   u(NX, 0:NY, 0:NZ)[nx, 1:ny-1, 1:nz-1] =
7                u(0, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
8   if (NX > 0)
9      u(1:NX, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] =
10      u(0:NX-1, 0:NY, 0: NZ)[nx-1, 1:ny-1, 1:nz-1];
11  u(0, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] =
12           u(NX, 0:NY, 0:NZ)[nx-1, 1:ny-1, 1:nz-1];
```

Figure 22. Code for explicit shadow region exchange in dimension x in the original HTA program for MG

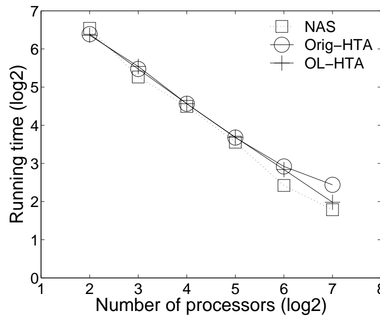


Figure 23. Performance of parallel MG benchmark class C

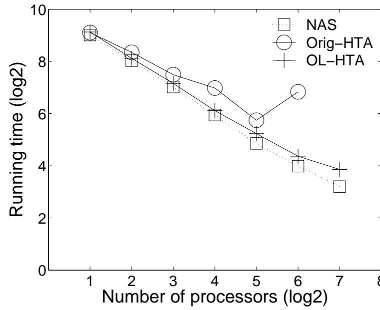


Figure 24. Performance of parallel LU benchmark class B

gram, more source lines of codes were written to deal with communications between neighbors than for computations. In an HTA code without overlapped tiling, the lines of code for communication of shadow regions are greatly reduced with respect to other programming approaches, but are still complex. Figure 22 shows the communication incurred in the dimension x in the original HTA code. In order to perform a correct update for a single dimension, the programmer has to write 48 indexing operations and derive the correct range for each indexing. With the overlapped tiling construct, the codes handling data exchange of the shadow regions are completely removed from the program.

Figure 23 shows the performance for three versions of the MG benchmark in class C: the MPI NAS benchmark (NAS), the original HTA program without overlapped tiling (Orig-HTA) and the HTA with overlapped tiling (OL-HTA). It shows that the new language construct performs better than the original HTA version because it avoids the indexing operations shown in Figure 22. The performance of OL-HTA is also close to that of the NAS benchmark.

The Symmetric Successive Over Relaxation (SSOR) algorithm used by LU represents a different class of stencil computation which contains loop-carried cross-processor data dependences that serialize computations over distributed array dimensions. Such computations are called *pipelined* [18]. We use a wavefront algorithm [7] which iterates the tiles in a special order along the diagonal (or hyperplane for high dimensions).

The explicit shadow region updates in wavefront algorithms are complex because tiles in different positions need updates from different neighbors. Our original HTA implementation of LU used a set of 54 indexing operations to perform this update in each iteration. On the other hand, with overlapped tiling, the complex indexing can be removed. The automatic update mechanism, disabled when appropriate, allows our data type to efficiently update the shadow regions with minimal number of communications.

The performance of LU in class B is presented in Figure 24. The three versions are the same ones mentioned in the MG benchmark. We show that the overlapped HTA program outperforms the original HTA program since it eliminates the complicated indexing operations. Its performance is similar to that of the NAS version.

6.3 Readability and productivity improvement

In this section, we summarize the impact of using the HTA data type on the ease of programming. Although there is no direct formula to evaluate the readability of programs, we use three metrics for sequential programs: the programming effort [17], the cyclomatic number [22], and the source lines of code.

In software science [17], a program is considered a string of tokens, which can be divided into two groups: *operators* and *operands*. The *operands* are variables or constants. The *operators* are symbols or combinations of symbols that affect the value or ordering of operands. We denote η_1, the number of unique operators; η_2, the number of unique operands; N_1, the total occurrences of operators; and N_2, the total occurrences of operands. The program level *Lvl*, and the program volume *Vol* are defined as functions of η_1, η_2, N_1 and N_2 (see [17]). The programming effort E can be obtained using the estimate of *Lvl* and *Vol* as E $=$*Lvl*$/$*Vol*. It is assumed that the programming effort required to implement an algorithm is proportional to E.

The second metric, the cyclomatic number [22], is a function of the number of predicates in the program. In particular, if a control flow graph (CFG) represents a program containing P decision points or predicates, then the cyclomatic number $V = P + 1$. Again, the smaller the V, the less complex the program is.

The third metric counts all the source lines L in the code ignoring the comments and empty lines, as we also want to produce as few lines of code as possible.

Table 1 shows the measurements for the three metrics for the three LU algorithms discussed in the paper and an algorithm to solve the triangular sylvester equation [25]. According to these metrics, the HTA programs require the least programming effort, and the lowest cyclomatic number. The HTA programs also require the least number of lines. One of the interesting observations is that all the non-HTA programs have large counts for the operator ",". For example, "," appears 65 times in NDS, 70 times in LAPACK, 95 times in the FLAME API and 74 times for the ATLAS code, while 33 times in average in HTA. In most cases, "," is used as a separator in function interfaces. This indicates that HTAs simplify the interface for functions. We also notice that HTA programs have a very small portion of arithmetic operations, which are usually used for index calculation. For example, operators `+, -, =` have 101 counts for the iterative algorithm using NDS but only 12 counts for HTA. This reveals the simplification of indexing using the HTA data type.

The biggest improvement for parallel programs is the simplification or removal of communications derived from overlapped tiling. Further, the indexing scheme this construct provides reduces

Programs	η_1	N_1	η_2	N_2	E	V	L
staticLU HTA	17	78	13	65	61,545	1	10
staticLU NDS	16	182	26	193	208,074	7	49
staticLU LAPACK	28	176	20	142	160,509	6	37
dynamicLU HTA	16	89	12	72	51,599	1	13
dynamicLU FLAME	22	167	34	113	170,477	1	52
recLU HTA	21	88	15	72	85,530	2	18
recLU ATLAS	33	194	23	136	186,891	14	40
Syl HTA	23	323	41	272	423,404	6	47
Syl FLAME	25	375	82	270	700,629	6	95

Table 1. Summary of three metrics: programming effort (E), the cyclomatic number (V) and the source lines of codes (L).

Metric	Benchmark			
	MG Orig	MG OL	LU Orig	LU OL
Communication Statements	117	4	56	4
Indexing Operations	265	117	428	26

Table 2. Code difference between original (Orig) and overlapped (OL) HTA programs

the number of indexings. For example, arithmetic operations and assignments apply to the owned region of each tile. Using overlapped tiling, the reference to an HTA `h` means all the elements in the owned regions participate. On the contrary, an HTA program with explicit shadow regions has to index explicitly the inner regions for each tile. Table 2 illustrates the reduction of communication statements and indexing operations for LU and MG.

7. Related work

Tiling [20] has been extensively studied and used as a parallel execution enabler/optimizer [26] and as a mechanism to improve locality [32]. However, much more attention has been paid to the code transformations related to it and the estimation of the optimal tile sizes for different purposes, than to the integration of tiles and their manipulation in programming languages. Most programming environments do not provide any support for tiles (FORTRAN, C, ...), thus leaving all the work to the programmer. While there are some compilers that apply tiling automatically, the analyses required are non-trivial, particularly when dealing with parallel programs. Thus, compilers transformations are mostly limited to the exploitation of locality in affine codes [21]. Besides, it is often the case that the user needs to address the tiles directly to express an algorithm [5]. The result of the relevance of tiling and the lack of straightforward support for it in the programming languages has been the generation of programs that trade efficiency for code bloating and lack of readability to several degrees.

Most efforts to express tiles in a language restrict themselves to tell the compiler that an array is to be tiled in a certain way in order to be distributed, but do not allow the manipulation of those tiles. This is the case of the HPF [19] data mapping directives for block and block-cyclic distribution or UPC shared arrays [10]. Others, like Co-Array Fortran [24] let the user refer to the tiles and portions of them, but their co-arrays are subject to many limitations: they can only appear in parallel computations, all the tiles must have the same size, automatic co-arrays are not permitted, no mechanisms for recursive/hierarchical tiling or dynamic partitioning exist, etc. These limitations preclude the effective usage of tiles and can be found in other environments like POOMA [28]. Sequoia [12] deserves special mention, as it is the only approach where tiling is not data-centric. Instead its main focus is task decomposition; thus Sequoia works on a hierarchy of tasks rather than a hierarchy of arrays. This way concepts such as the overlapped tiling introduced in this paper are foreign to Sequoia.

Finally, FLAME [6] provides APIs to help express its tile-based algorithms in different languages. While these APIs emphasize dynamic partitioning, our experiments suggest that HTAs are a more powerful way to express this kind of algorithms. The main reason is the large number of arguments of some FLAME functions, which lead to large invocations and the usage of many variables. HTAs avoid this at the cost of using more indexing operations.

8. Conclusions

In conclusion, tiled algorithms have received considerable attention as they help to exploit parallelism as well as optimize data locality. However, most programming languages lack language constructs related to tiling and place the burden of manipulating tiles on the programmer.

In this paper we have proposed the HTA data type, as a natural extension of current OO languages to facilitate tile programming. Using HTAs, algorithms can be naturally described in terms of tiles, the most appropriate data layout can easily be chosen, and tile management is greatly automated.Consequently HTAs effectively reduce the extra indexing calculations, simplify the function interface and particularly reduce the complexity of parallel programming, while not having a negative performance impact. Two new language constructs, namely dynamic partitioning and overlapped tiling, that make tiling more general and flexible, have been proposed. Finally, our experimental results demonstrate the effectiveness of the proposed approach.

Acknowledgments

We thank Christoph von Praun and James Brodman for their contributions to this research. This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432. Basilio B. Fraguela was partially supported by the Ministry of Education and Science of Spain, FEDER funds of the European Union (Projects TIN2004-07797-C02-02 and TIN2007-67537-C03-02).

References

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

[2] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Trans. Comput.*, 30(5):341–356, 1981.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral. Shared Memory Programming for Large Scale Machines. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–117, 2006.

[5] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw.*, 31(1):1–26, Mar. 2005.

[6] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn. Representing linear algebra algorithms in code: the FLAME application program interfaces. *ACM Trans. Math. Softw.*, 31(1):27–59, 2005.

[7] G. Bikshandi. *Parallel Programming with Hierarchically Tiled Arrays*. PhD thesis, UIUC, 2007.

[8] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *PPoPP '06: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 48–57, 2006.

[9] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. B. Fraguela, M. J. Garzarán, D. Padua, and L. Rauchwerger. Design and use of htalib - a library for Hierarchically Tiled Arrays. In *Proc. of the Intl. Workshop on Languages and Compilers for Parallel Computing*, 2006.

[10] 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.

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[12] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Supercomputing '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83, 2006.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems*. Prentice-Hall, Inc., 1988.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, 1999.

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

[16] F. G. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev.*, 47(1):31–55, 2003.

[17] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[18] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 86–100, 1991.

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

[20] F. Irigoin and R. Triolet. Supernode Partitioning. In *POPL '88: Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 319–329, 1988.

[21] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *PPoPP '01: Proc. of the 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 103–112, 2001.

[22] McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[23] A. C. McKellar and J. E. G. Coffman. Organizing Matrices and Matrix Operations for Paged Memory Systems. *Communications of the ACM*, 12(3):153–165, 1969.

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

[25] E. S. Quintana-Ortí and R. A. van de Geijn. Formal Derivation of Algorithms: The Triangular Sylvester Equation. *ACM Trans. Math. Softw.*, 29(2):218–243, 2003.

[26] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 111–120, 1991.

[27] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[28] 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 *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[29] S. Toledo. Locality of Reference in *LU* Decomposition with Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[30] T. Veldhuizen. Techniques for scientific C++. Technical Report TR542, Department of Computer Science, Indiana University, 2000.

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

[32] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 30–44, 1991.

[33] M. Wolfe. More Iteration Space Tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.